GameBoy Advance Programing Bible

ゲームボーイアドバンス・プログラミングバイブル

●オリジナルゲームとUSB転送ツールを作ってGBAで遊ぼう！

CD-ROM付属

携帯ゲーム機用プログラムのクロス開発に挑戦しよう！

●前半は…？

ＧＢＡの動作原理を考えながら、その上で動作するプログラムをとおしてプログラムの組み方を憶えていく方法をとります。

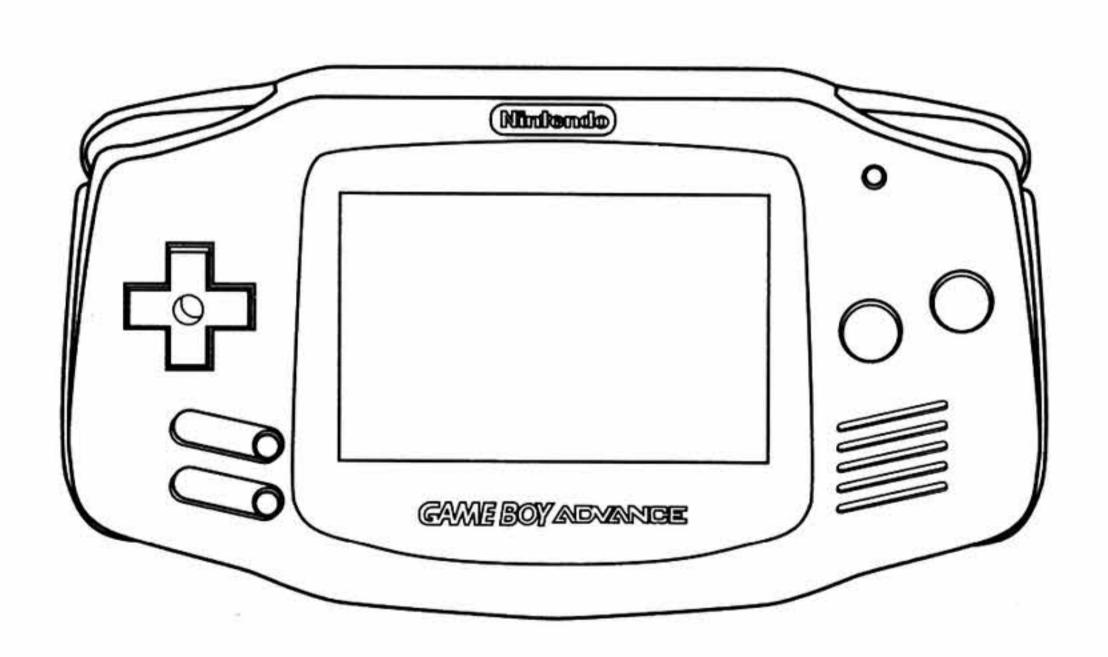
●後半は…？

実際にビルドしエミュレータで動作確認したプログラムを、ＧＢＡに転送して実機動作に挑戦します。ここでは、プログラム転送ツールを自作し、付属プログラムでＧＢＡをＰＣでコントロールする、あるいはその反対を体験できます。

GameBoy Advance Programing Bible

CD-ROM
付属

大橋 修
Ohashi Osamu

Introduction

PC以外で動くプログラム作成にチャレンジしてみよう!

■PCで動作するプログラムにチャレンジした次のステップは？

日本が世界に誇る文化は旧来の伝統芸能に加えて、アニメやゲームだそうです。この2つは最近でこそ注目されていますが筆者の子供の頃は親から目の敵にされていたものでした。変われば変わるものです。

ゲームといっても、現在はテレビゲームをはじめとしたコンピュータを使ったゲームが主流です。とくに携帯電話の普及にともない、携帯機器で楽しむゲームが市民権を得るようになりました。電車に乗れば大人は携帯電話で落ちモノパズルをやっていますし、子供は本書でも紹介する任天堂のゲームボーイシリーズ（カラー／アドバンスなど）で遊んでいる光景をよく見かけます。

ところで、昨今のITブームの効果なのか、PC用のプログラムを組める人は増えているようです。しかし、それに対してPC以外の環境ではさっぱりというのも現実です。ユビキタスや情報家電と呼ばれるPC以外のコンピュータも増えてきましたし、秋葉原やインターネットでは、それらを制御している組み込み型のマイコンを購入することさえできるのにです。

本書はPC以外の機器で動作するプログラムを組むためのきっかけづくりを提供する本です。もちろんPC以外といっても、部品剥き出しの組み込み型マイコンをいきなり見せ付けられても、その迫力に少し引いてしまいます。なんか、ちょっと怖いですよね？

本来これらはコンピュータを意識せずに使えるのが売りのはずですが、ここでは敢えて意識したいと思います。そこで、もっとも身近なユビキタス（なんか変な言葉だな？）である、携帯ゲーム機、その中でも最新のマシンであるGameBoy Advance（ゲームボーイアドバンス）のプログラミングにチャレンジします。GameBoy Advance（以下GBAと略）の動作原理を考えながら、その上で動作するプログラムをとおしてプログラムの組み方を憶えていく方法をとります。

携帯型ゲーム機の最大の魅力は気楽に携帯できることです。かっこいいプログラムができたら、みんなに見せて自慢してください。そこから、ひとりでも多くの人がコンピュータの原理や魅力を発見できるきっかけとなれば、筆者としては望外の喜びです。

■言い訳

日本が世界に誇る分野だけあって、その世界は深遠です。本書はほんのさわり程度でしかありません。たぶんGBAの実力の100分の1も引き出していないのではないかと思います。ただ、100分の1程度でもここまでできるということを、読者の皆さんには理解していただきたいと思っています。短く簡単なプログラムでもこんなに遊べる、それではもっと頑張ったら……。そこから先は読者の皆さんにお任せいたします。クールなプログラム、システムをジャンジャン作って、完成したらどうぞWebで公開してください。

■本書の範囲

いままでにPCでC言語などを使ってプログラムを組んだ経験があり、「HelloWorld!」程度までは完了している読者を想定しています。まずはPC用のC言語の環境で、コンパイラやエディタなどの開発ツールをひととおり使ってみることをオススメします。

また、本書の中心になるのはプログラムを読むことです。本書で取り上げているプログラムはオープンソースとして開発されていることもあり、可読性の高いプログラムになっています。プログラムの組み立て方と同時に、プログラムの書き方も参考にしてほしいと思います。

本書の後半は実際にビルドし、エミュレータで動作確認を行なったプログラムをGBAに転送して実機動作にチャレンジします。ここでは、プログラムの転送ツール（ハードウェア）を自作し、付属のプログラムを使ってGBAをPCでコントロールする、あるいはその反対にGBAでPCをコントロールすることを経験できます。つまり、クロス開発を最初から最後まで体験できるわけです。

■商標

本書および付属CD-ROM内の各ファイル／バイナリ等に掲載されている会社名、システム名、ソフトウェア名、製品名等は一般にその開発元、発売元の商標または登録商標です。また、本文中でとくに断り書きのない場合、GBAはGameBoy Advanceを、GBCはGameBoy Colorを表わします。なお、本文中では、™、®のマークは明記していません。

■注意

ULA等の転送ツールは、本書の内容を充分に理解した上で、自己責任において製作してください。製作不良（ハンダづけ不良、配線ミス等）によって各種ツールの回路破壊、GameBoy Advance本体が故障したり障害等が発生しても、筆者および出版社は一切責任を負うものではありません。

また、本書に付属するCD-ROMに収録したソフトウェアおよびプログラムを許可なく再配布およびWeb等で公開することを固く禁じます。

GAME BOY ADVANCE PROGRAMING BIBLE

もくじ

CONTENTS

Chapter 01 GameBoy Advanceの生い立ち

1-1 ケータイゲーム機戦国記 ···· 012
1-2 GBAのしくみ ···· 014
 1-2-1 スペックの読みこなし ···· 016

Chapter 02 開発ツール（初級者・中級者向け） Development Tools

2-1 ソフトウェア開発（クロス開発のおきて） ···· 022
2-2 ソースコードと実行コード ···· 024
 2-2-1 ビルド ···· 026
 2-2-2 開発ツールとは ···· 027
2-3 開発ツール紹介 ···· 028
 2-3-1 GBACC ···· 028
 2-3-2 DevkitAdv ···· 029
 2-3-3 ペイントブラシ ···· 030
 2-3-4 エミュレータ ···· 031
 2-3-5 グラフィックコンバータ ···· 033
 2-3-6 その他のツール ···· 034
2-4 開発ツールのインストール ···· 035
 2-4-1 開発ツールのパス設定 ···· 036
  2-4-1-1 GBACC ···· 036
  2-4-1-2 VBA ···· 037
  2-4-1-3 bmpcnv ···· 037
  2-4-1-4 AGBGFX ···· 037
 2-4-2 開発ツールの動作確認 ···· 039
  2-4-2-1 GBACCとbmpcnvの動作確認 ···· 039
  2-4-2-2 VBAの動作確認 ···· 041
  2-4-2-3 ペイントの動作確認？ ···· 041

Contents

Chapter 03 GameBoy Advance プログラミングⅠ
ゲーム制作でGBAのプログラミングを学ぶ

3-1 GBAでゲームを作る必然性……044
3-2 大規模なソフト??……045
3-2-1 CPUパワーを使うソフト？……045
3-3 リバーシ（オセロ）ゲームの制作（BONSAI－WARE）……046
3-3-1 ゲームの構成……046
3-4 ゲームの構成要素の分析……048
3-4-1 文字を出す……048
3-4-2 画面に絵を出す……050
3-4-2-1 VRAM……050
3-4-2-2 VRAMへの書き込みと画面出力……051
3-4-2-3 1画面分の表示……051
3-4-2-4 AGBGFXの使い方……052
3-4-3 ゲームパッドの読み取り……054
3-4-4 カーソルの移動……055
3-4-4-1 カーソルの形状……055
3-5 画面の切り替え（場面設定）……056

Chapter 04 GameBoy Advance プログラミングⅡ

4-1 プログラムの構造（メイン）……060
4-2 各構成要素の詳細分析……065
4-2-1 オープニング……065
4-2-1-1 オープニングの要求分析……065
4-2-1-2 ビットマップファイルの表示……066
4-2-1-3 スタートキーが押されたらゲーム開始……066
4-2-1-4 プログラムを書いてみる……067
4-2-2 色設定……075
4-2-2-1 色設定の要求分析……075
4-2-2-2 オープニングとの違い……075

Contents

4-2-2-3　プログラムを書いてみる……076
4-2-3　レベル設定……081
4-2-3-1　レベル設定の要求分析……081
4-2-3-2　色設定との違い……082
4-2-3-3　プログラムを書いてみる……082
4-2-4　ゲーム……083
4-2-5　レンダラー……084
4-2-5-1　レンダラーの要求分析……084
4-2-5-2　画面構成……085
4-2-5-3　プログラムを書いてみる……085
4-3　状態遷移……106
4-4　問題点の確認……107
4-4-1　容量とスピードのトレードオフ……107
4-4-2　データ圧縮テクニックとプログラムの実装……108
4-4-2-1　BmpCnv……108
4-3-3　makefileとmake1.batの作成……111
資料　ReversiAdvanceのエンジン……114

Chapter 05 **自作ライブラリの説明**
TeamKNOxLib

5-1　GBAプログラミングでは独自のライブラリ構築が不可欠！……128
5-2　ライブラリの分類と使い方……129
5-2-1　グラフィックライブラリ……129
5-2-2　通信ライブラリ……131
5-2-3　文字列操作ライブラリ……132
5-2-4　乱数ライブラリ……133
5-2-5　ボタン操作ライブラリ……133
5-2-6　時間ライブラリ……134
5-2-7　その他のライブラリ……134
5-2-8　グローバル変数……134

Contents

5-3 ソースコード ········ **135**
5-3-1 TeamKNOxLibの仕組み ········ **135**
5-3-2 TeamKNOxLibのこれから… ········ **135**

Chapter 06

GameBoy Advance 実機動作（中級者・上級者向け）
Actual Device Demonstration

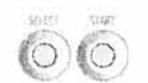
6-1 実機動作の必要性 ········ **152**
6-2 GBA実機動作の歴史 ········ **153**
6-3 xLAプロジェクト(PLA, ULA) ········ **156**
6-4 実機動作の原理 ········ **157**
6-4-1 動作シーケンス ········ **157**
6-4-1-1 全体図 ········ **157**
6-4-1-2 ファームウェアのダウンロード ········ **158**
6-4-1-3 1.5 BIOSのダウンロード ········ **158**
6-4-1-4 2nd BIOSのダウンロード ········ **159**
6-4-1-5 通信開始 ········ **159**
6-5 ULAの製作 ········ **160**
6-5-1 初心者向け(AN2131SC [EZ-USB] 評価ボードを利用) ········ **161**
6-5-1-1 IPI社製評価ボード ········ **161**
6-5-1-2 Mini-EZ-USB (EZ－ULA) ········ **163**
6-5-2 上級者向け(市販製品の改造) ········ **165**
6-5-2-1 USB-PDC ········ **165**
6-5-2-2 CE-PD03 ········ **167**
6-5-3 改造方法 ········ **168**
6-5-3-1 2121で動作した!? ········ **168**
6-5-3-2 で、ワンコイン… ········ **168**
6-5-3-3 実際に作ってみると… ········ **169**
6-6 自作プログラムの転送 ········ **174**
6-6-1 ULA-HostV2 ········ **175**
6-6-2 基本的な操作 ········ **176**
6-7 開発者向けの機能 ········ **179**

Contents

6-8 その他のツールたち(ULA以外の第4世代のツール)……180
6-8-1 F2A-USB Linker……180
6-8-2 ブートケーブルUSB……181
6-8-3 ULA-FX……182

Chapter 07 GameBoy Advance 活用事例(ユーザー主導)
Case of Exploitation

7-1 FlashManager……186
7-2 GBAのBIOSの吸い上げ……187
7-2-1 BIOSって?……187
7-2-2 BIOSの吸い上げプログラム……188
7-2-3 BIOS吸い上げの原理……189
7-2-3-1 バイナリエディタ……190
7-2-4 BIOSの吸い上げ実践編……191
7-2-5 BIOSの設定……195
7-2-6 BIOSの確認……195
7-3 GBAをPCのゲームパッドにする……196
7-3-1 ULA-GP……196
7-3-2 ULA-GPの動作原理……197
7-3-2-1ULA-GPの動作シーケンス……197
7-3-3 ULA-GP_V2……198
7-3-3-1 ULA-GP_V2の特徴……198
7-3-3-2 ULA-GP_V2の動作シーケンス……199
7-4 GBAでファミコンエミュレータを楽しむ……202
7-4-1 ファミコンエミュレータ……203
7-4-2 Thingy-ULA……204
7-4-3 Thingy-ULAの使い方……207
7-4-3-1 基本的な使い方……208
7-4-3-2 拡張部分の使い方……211

Contents

Chapter 08 TeamKNOx

8-1 TeamKNOx 活動の歴史 …… 214
8-2 ビジネスモデルとプレイモデル …… 216
8-2-1 楽しくやる …… 217
8-2-2 お金をかけない …… 217
8-2-3 共有する …… 217
8-3 TeamKNOx の Web ページはなぜ英語なのか？ …… 218
8-4 最近、気になっていること …… 219
8-4-1 フリーウェアとオープンソース …… 219
8-4-2 ゲーム脳と手作りゲーム …… 219
8-4-3 日本の未来 …… 221

最後に …… 222
付属 CD-ROM について …… 223

Column

● ARM プロセッサとは …… 020
● クロス開発秘話 …… 042
● GCC と Cygwin …… 042
● チェスとリバーシ …… 055
● トレードオフ …… 058
● BONSAI － WARE とは？ …… 058
● カーソルの描画情報の読み方 …… 080
● 1つ前のボタンの情報が必要なわけは？ …… 109
● ヘッダーファイルを使うわけ …… 126
● ハンダづけのコツ …… 182
● ULA の入手方法 …… 183
● 電子工作について …… 183
● エミュレータの再現性 …… 184
● カートリッジの吸い上げ …… 212

Game Boy Advance の生い立ち

まず最初に、これからプログラムの開発を行なうゲームボーイアドバンスという、この魅力的なマシンについて簡単に触れておきましょう。
ここでは公開されているスペックから、その潜在能力を探っていきます。

ケータイゲーム機戦国記

初代ゲームボーイが発売されたのは1989年です。その成功を見て、各社からさまざまな携帯ゲームが発売されました。お手軽にどこでもゲームを楽しめ、ソフトの開発コストも抑えられる携帯型ゲーム機をメーカーが放っておくわけがありません。「100Mショック・ネオジオ」のネオジオポケットをはじめ、ゲームボーイの開発者である故・横井軍平氏の手によるワンダースワンは1999年の3月に発売されました。ただ、まわりを見回すとわかりますが、ゲームボーイほどの成功はおさめていないのが実情だと思います。

ゲームボーイ	1989年 4月
ゲームボーイポケット	1996年 7月
ゲームボーイライト	1998年 4月
ゲームボーイカラー	1998年10月
ゲームボーイアドバンス	2001年 3月
ゲームボーイアドバンスＳＰ	2003年 2月
ワンダースワン	1999年 3月
ワンダースワンカラー	2000年12月
ワンダースワンカラークリスタル	2002年 6月
ネオジオポケット	1998年10月
ネオジオポケットカラー	1999年 3月（廉価版は10月）

携帯型ゲームは子供が遊ぶものという概念を2003年2月に発売された、ゲームボーイアドバンスSPは見事に打ち破りました。昨今のケータイの主流のデザインである折り畳み型を大胆に取り入れ、大人が遊んでいても違和感がありません。新規参入してくる子供たちは当然ですが、卒業してしまった大人たちをも取り込んでしまっています。

ケータイゲーム機にも歴史あり

年	機種
1989年	★GameBoy
1990年	
1991年	
1992年	
1993年	
1994年	
1995年	
1996年	★GameBoy Pocket
1997年	
1998年	★GameBoy Light ★NEOGEO Pocket ★GameBoy Color
1999年	★Wonder Swan ★NEOGEO Pocket Color
2000年	
2001年	★Wonder Swan Color ★GameBoy Advance
2002年	★Swan Crystal
2003年	★GameBoy Advance SP
・ ・ ・ ・	

ゲームボーイ　ネオジオポケット　ワンダースワン

Chapter 01

GBAのしくみ

GBAの大まかなスペックは任天堂のカタログやWebで確認することができます。以下、Webからの抜粋です。GBAも含めてゲーム機とは洗練された入出力（きれいなLCD、押しやすいキーパッドなど）を持った組み込みシステムには違いありません。コンピュータとして見たときのこれらのスペックの裏側にあるものを探っていきましょう。

GameBoy Advance

GameBoy Advance SP

商品名	ゲームボーイ アドバンスSP	ゲームボーイ アドバンス
液晶	2.9 inch 反射型TFTカラー液晶 （フロントライト付）	2.9 inch 反射型TFTカラー液晶
画面サイズ	40.8mm × 61.2mm	
解像度	240 × 160 ドット	
表示能力	32000色	
CPU	32bit RISC－CPU ＋ 8bit CISC－CPU	
メモリ	32KB WRAM ＋ 96KB VRAM（CPU内蔵） 256KB WRAM（CPU外部）	
サウンド	スピーカー、ヘッドホン端子付 ※1	
通信機能	ゲームボーイアドバンス専用通信ケーブルによる4人までのマルチプレイが可能	
使用電源	内蔵リチウムイオン充電池	単3形アルカリ乾電池2本、専用バッテリーパック、専用ACアダプタセット
電池持続時間	ライトON時：約10時間（フル充電時） ライトOFF時：約18時間（フル充電時） 充電時間：約3時間 （空の状態からフル充電した場合）※2	単3形アルカリ乾電池：約15時間 専用バッテリーパック：約10時間
最大消費電力	約1.6W（充電時）	約0.6W（ゲームプレイ時）
寸法	縦84.6mm × 横82mm × 厚さ24.3mm（折りたたみ時）	縦82mm × 横144.5mm × 厚さ24.5mm
本体重量	約143g（バッテリーパックを含む）	約140g（乾電池含まず）
対応ソフト	ゲームボーイアドバンス専用カートリッジ／ゲームボーイカラー対応カートリッジ／ゲームボーイカラー専用カートリッジ／ゲームボーイ用カートリッジ	

※1 ゲームボーイアドバンスSPでヘッドホンを使う場合はヘッドホン変換プラグが必要です。
※2 充電池の寿命は使用頻度によって異なりますが、500回の充放電後の使用時間は初期の約70%になります。

スペックの読みこなし

スペックが読みこなせるようになると、機能の確認が簡単に行なえるようになります。また、他の機械との比較なども容易に行なえるようになります。

■ LCD（Liquid Crystal Display＝液晶ディスプレイ）

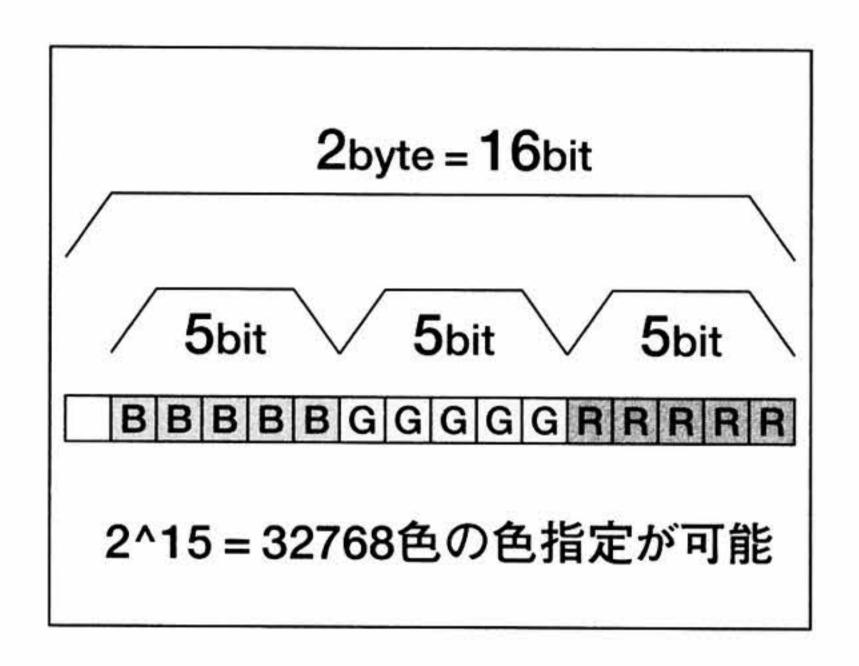
2.9インチはLCDの対角線の長さを表わします。この辺はTVと同じですね。反射型はバックライトなどがないシステムです。TFTはThin Film Transisterの略称です。これはLCDの動作方式の一種で、その他にSTN（Super Twisted Nematic）などがあります。TFTはSTNなどの他の方式に比べて一般に動作が速く高性能とされています。ゲームボーイ周辺に限るとゲームボーイカラーではじめて採用されました。カラー液晶は文字通りカラーの液晶のことですね。※1inchは約2.54cm

■ 解像度

解像度は横240ドット×縦160ドットです。いままでのゲームボーイの160×144に比べて大幅に増強され、ソフト（ゲーム）の表現力がアップしました。当たり前のことですが、これより大きな絵を表示する場合は縦横を圧縮するなり、スクロールさせるシステムが必要になります。

■ 表示能力

32000色となっていますが、これは32768色が表示可能な色数です。コンピュータで色を出す方法は、RGBのデータの重み付けで表現します。32768は2の15乗（2^{15}）になります。これを15ビットなどと表現しますが、コンピュータに取り扱いやすい数は2のべき乗です。つま

り、「xRRRRRGGGGGBBBBB」のxにはデータはありません。RRRRRGGGGGBBBBBの部分に0か1が入ります。x000000000000000〜x111111111111111の数の取りうる範囲が0〜32767となります。ここで32767となるのは0自身も色として含まれるからです。32768個つまり、32768色となります。

■ CPU

GBAが発表された当初、GBAは任天堂では初の下位互換性を保持したゲームマシンとして注目されました。つまり、いままでのゲームボーイのソフトも動作するわけです。これが「8bit CISC−CPU」の部分になるわけです。「32bit RISC−CPU」の部分はARM−CPUが使われています。筆者自身、情報収集のために各種の展示会などに足を運びます。CPU関連の展示会でARMブースではGBAそのものをARMを使っているマシンとしてデモを行なっていました。

■ メモリ

スペックを見ると「CPU内蔵」と「CPU外部」となっています。CPU内蔵とは、文字通りCPUと共にワンパッケージにまとめられているメモリのことです。これはCPUの動作サイクルに近い速度で動作させること

ができるので処理速度の点では有利になりますが、同じパッケージ内のため容量的には大きく取れません（コスト高の要因になります）。

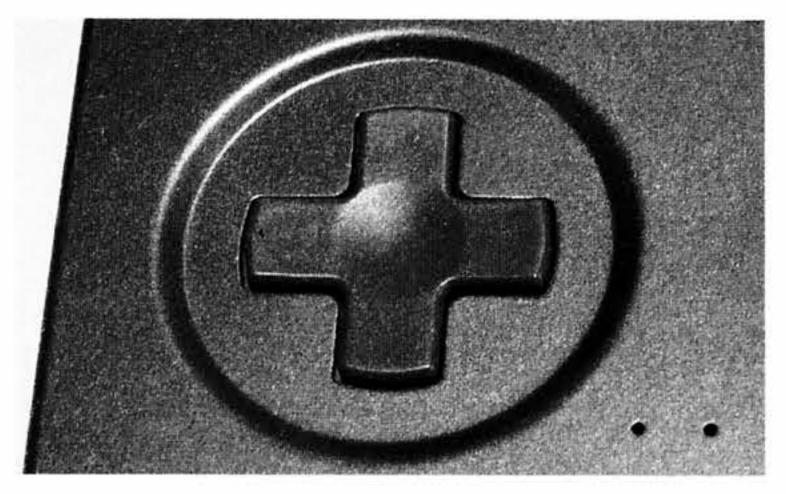
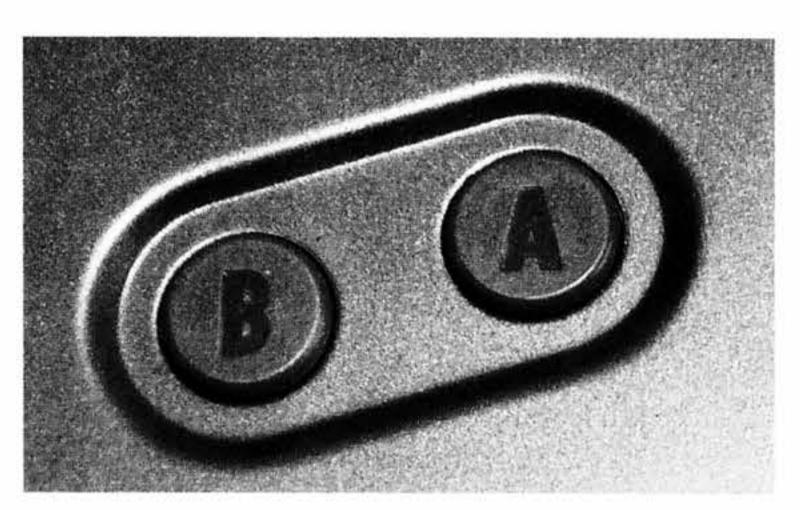
「32KB WRAM」とはプログラムが格納される領域です。「96KB VRAM」は画像表示用のメモリで、VRAMとはVideoRAMの略称です。昨今のハイエンドPCのディスプレイボードのVRAMでは64MB～128MBを積んでいることを考えると、GBAはハイエンドPCの1000分の1程度のVRAMということになります。

それでは次に96KBとはどういうことかを考えてみましょう。GBAの解像度は240×160ドット、色数は32768（2byte）であることを考えると、240×160×2＝76800となります。75KBあればスペックのとおりです。76KBにならないのはメモリ関係（コンピュータ関係？）では1024を1Kとおくためです。したがって76800÷1024＝75となります。さて、話を残りの21KBに戻すと、これら残りの分のメモリはゲームで使うパレットの領域や他の画面モード、オブジェクト（スプライト）に使われているようです。

■サウンド

スペックでは「スピーカー、ヘッドホン端子付」としか書いてありませんが、ゲームボーイのサウンド＋αと考えれば間違いないでしょう。インターネットなどの情報では、ダイレクトサウンドなどのメモリに蓄えた音声データを出力できるようです。

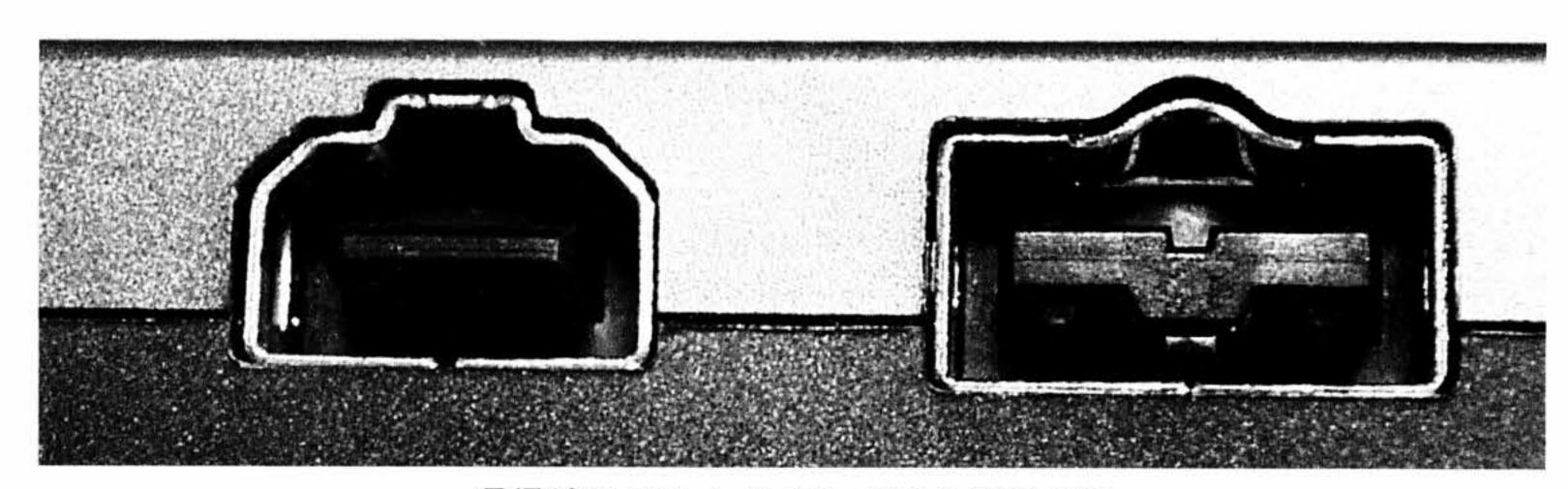
■ボタン

これはスペック表には出ていませんが、入力用の十字ボタンやA、Bボタンなども装備されています。GBからの進化としてL、Rボタンが装備されるようになりました。

GameBoy Advance SPの十字ボタン

GameBoy Advance SPのＡＢボタン

GameBoy Advance SPのＬボタン

GameBoy Advance SPのＲボタン

■通信端子

GBAを特徴付けるものとして通信端子があげられます。この機能の活用（ポケモンの交換）によりポケモンが一大ヒットを飛ばしたのは記憶に新しいところです。GBAではGBに比べてかなり強化されました。

通信端子（左）とサウンド出力端子（右）

Column

ARMプロセッサとは

ARMが注目されだしたのはAPPLE社のPDA・ニュートンに採用されたのがきっかけです。このARMは元々、BBCが放送していた教育番組の教育用のコンピュータのBBCマイクロの開発に端を発しています。このBBCマイクロの第1弾は6502が使われていました。その第2弾を作るにあたって設計者はさまざまなCPUを検討したようですが最適なものがとうとう見つけられず、そこで自ら設計を行なうことにしたわけです。新規に設計されるわけですから、しがらみなどはありません。結果、搭載されたのは少ない命令でしたが、簡潔であるがゆえにパワフルなCPUになりました。また、集積度が少ないので消費電力も抑えられています。このことから、ARMはモバイルデバイスに数多く搭載されるようになったのです。消費電力あたりの処理能力は群を抜いています。

開発ツール
Development Tools

初級者・中級者向け

GBAのプログラム開発もPCのソフトウェア開発も実質的には変わるところがなく、ソースコードを記述してビルドし、実行コードを得ます。
ここではそのために必要な開発手順、そして開発ツールとそのインストール方法を紹介します。

ソフトウェア開発 (クロス開発のおきて)

ここではGBAのソフトウェア開発に必要な各種道具の説明を行ないます。GBAに限らずコンピュータを動作させるためにはプログラムが必要です。PCなどのプログラムはPC自身が持つキーボードやマウスなどでプログラムを入力して組むことができるようになっています。プログラミング言語もCやJAVAを使うことは、この本を手に取っている人は既にご存知のことと思います。

このようにプログラムの開発ができる状態を「プログラム開発環境」と言います。ではプログラム開発環境、つまりキーボードやマウス、プログラミング言語を持たないGBAではどのようにプログラムを開発するのでしょうか？ 答えはプログラム開発環境（PCのことですね）でプログラムを作り、それをGBAに内蔵させることで実現します。このようなプログラムの開発スタイルを「クロス開発」と言います。

内蔵とはGBA自身がそのプログラムを参照できるかたちにすることです。ここでは対象をGBAに限りましたが、これは何もGBAだけに限ることではありません。みなさんが持っているケータイ電話でも同じことです。GBAとかケータイとか区別するのは面倒なので「ターゲット」という言葉で表現することもあります。「ターゲット」の対になる言葉は「ホスト」です。つまり、『「ホスト」の「プログラム開発環境」で「ターゲット」の「クロス」開発を行なう』という感じでしょうか？

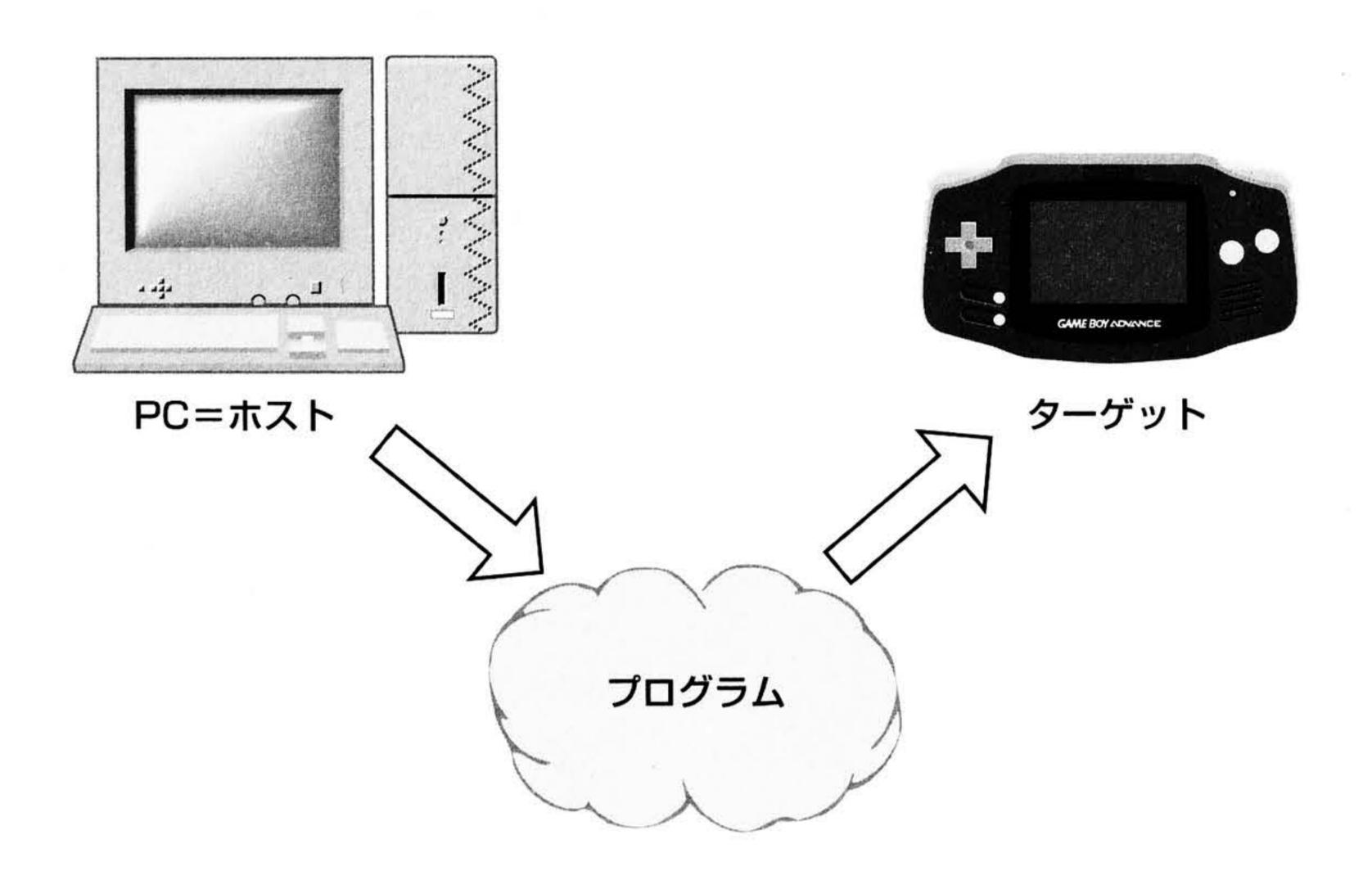
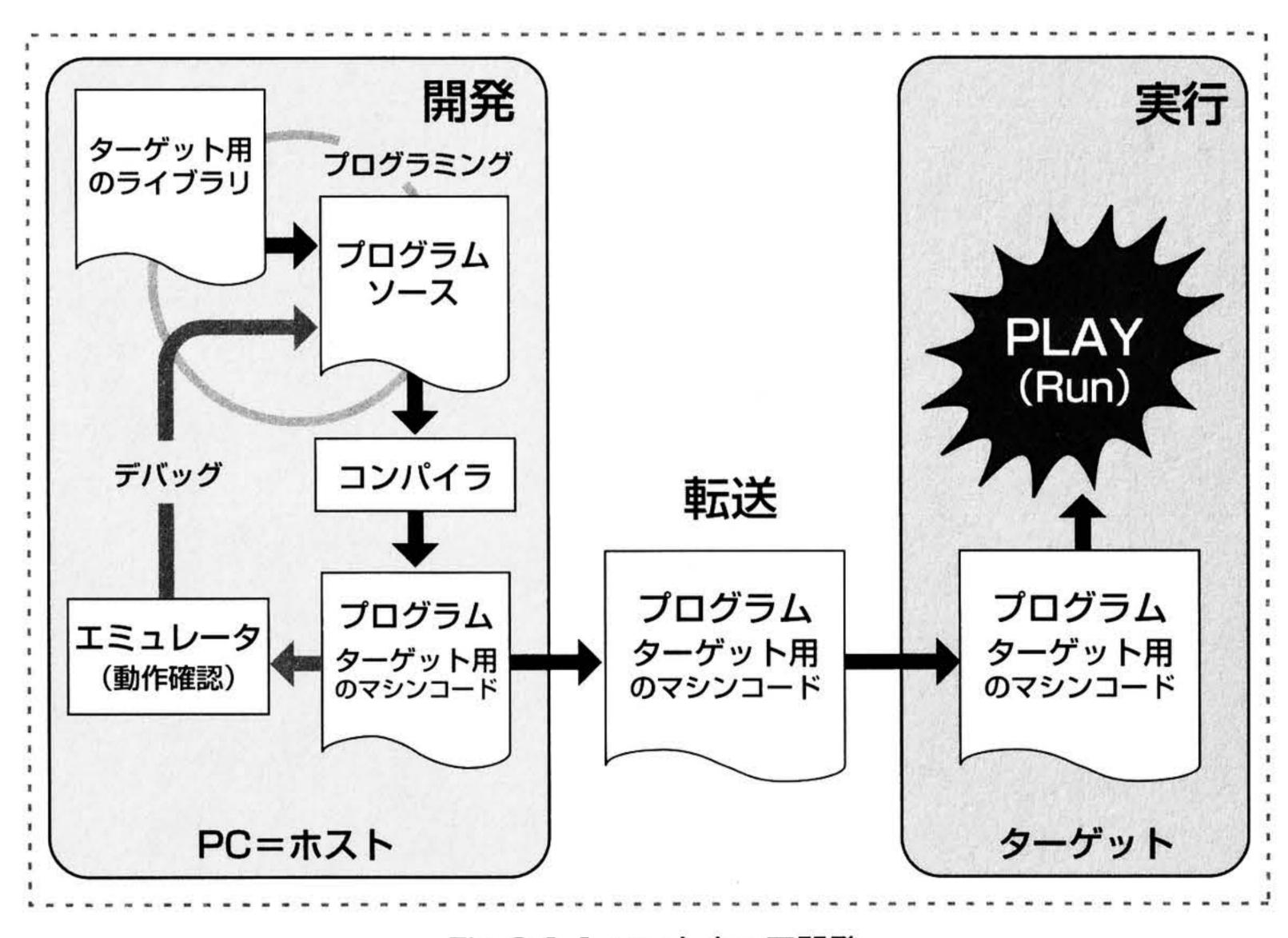



Fig 2-1-1 ソフトウェア開発

ソースコードと実行コード

ターゲットに限らず、すべてのコンピュータにおいてプログラムを実行するにはマシンコード(機械語とも言いますが最近はあまり使いませんね)が必要です。マシンコードは通常16進数になっています。これはコンピュータはメモリを参照してプログラムを実行していくので、そのメモリの格納形式を表わしているからです。

ところがメモリの格納形式をそのまま表わしたのでは一般の人には非常にわかりにくいものです。そこで人間にわかりやすいかたちでプログラム(ソースコード)を記述してそれをマシンコードに変換することにします。この作業を「コンパイル」と言います。このマシンコードに変換する作業はコンパイラが行ないます。これを手作業で行なうことをハンドコンパイルと言いますが、最近ではあまり見かけません。「Cのコンパイラ」というのは「Cのソースコード」を「マシンコード」(各CPU用コード)に変換するためのプログラムというわけです。

さて、それでは実行コードとは何でしょうか? 基本的には「マシンコード=実行コード」と考えていいと思います。ところが、JAVAなどでは少し様子が違います。JAVAの場合は、バイトコードと呼ばれるプログラムを実行します。これはちょうど、日本人として生まれて日本語の環境下で育った人が、その後習得した英語を話すようなものです(英語が喋れる日本人ですね)。つまり、最も効率よく話す(処理する)ことができるのは日本語ですが、とりあえずどこでも通用する、あちこちで話す(処理する)ことが

できるのは英語ということになります。処理速度を犠牲にして融通性を強化したものと考えることができます。見た目には英語で処理していますが、最終的には日本語で処理しているわけです。この場合、英語が「実行コード」で日本語が「ネイティブコード」になります。なんか昨今のグローバル化のようですね。JAVAが受けいれられるのはこのようなことも一端にあるのかもしれません。

Fig 2-2-01 ソースコードと実行コード

C言語

実行コード
（マシンコード）

OS

CPU

Java

実行コード（バイトコード）

ネイティブコード

バイトコードを実行

Java VM（Virtual Machine）

ネイティブコードを実行

CPU

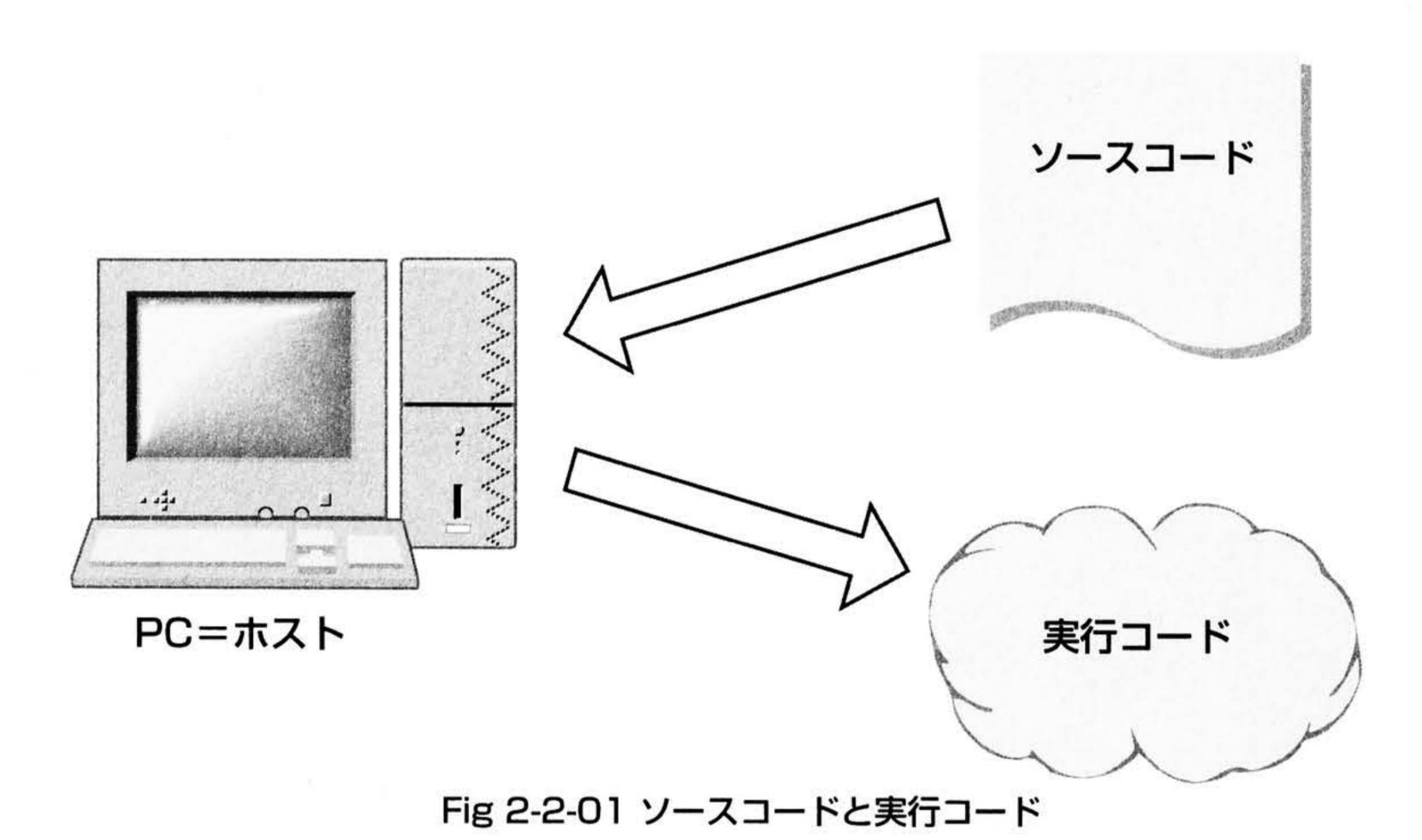



Fig 2-2-02 実行コードとネイティブコード

ビルド

　ビルドという言葉もよく聞きますね。最近のプログラムにはソースコードだけではなく絵（ビットマップ）や音（*.wav）なども含まれています。絵はペイントソフトで描いたり、音はWAVEファイルなどで用意します。プログラムとは直接関係ないこれらのファイルはリソースなどとも呼ばれますが、これらリソースを含めた実行コードを作成する行為をビルドと言います。

ソースコード
ソースコード
ソースコード
リソースファイル
PC＝ホスト
実行コード

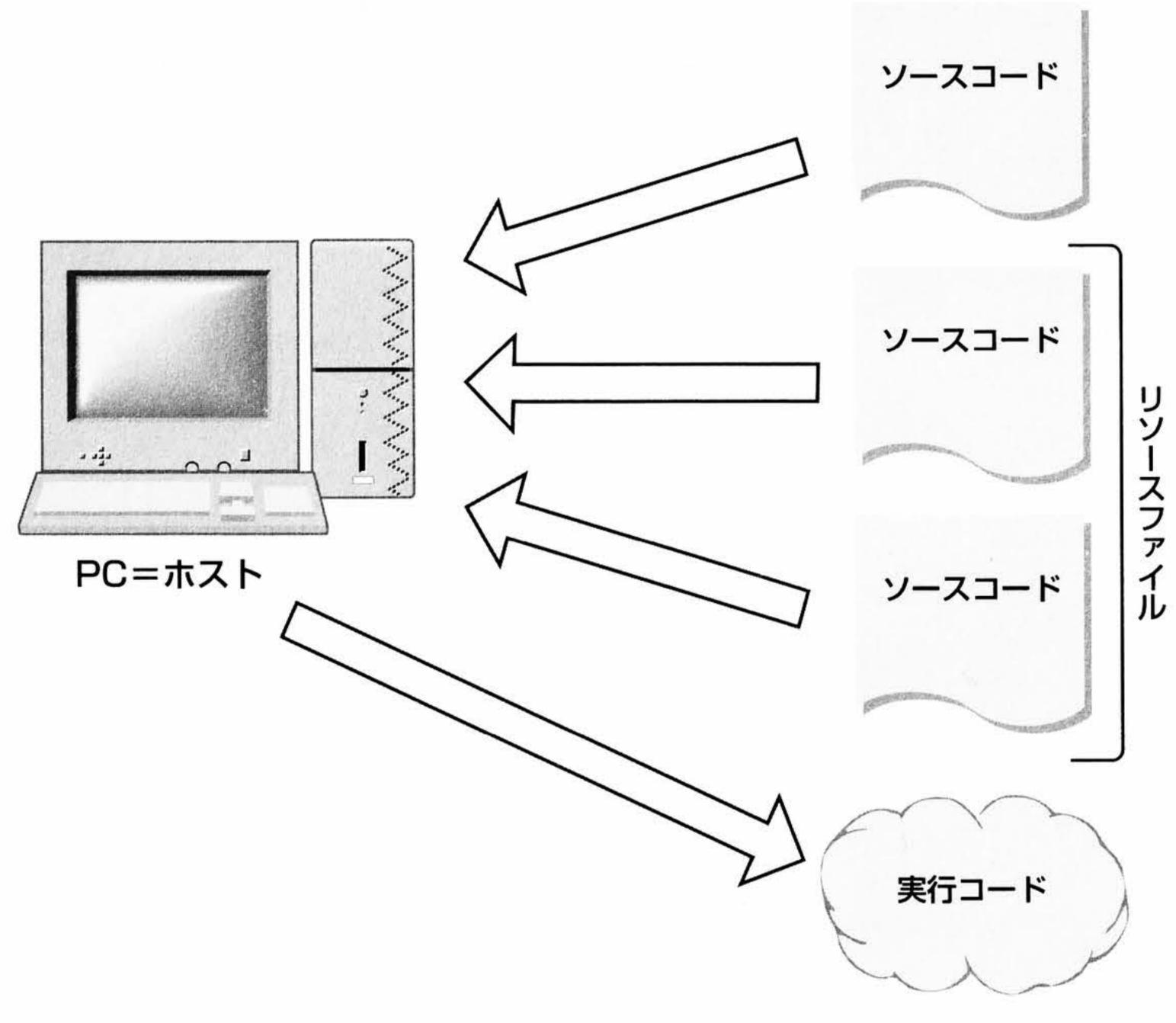



Fig 2-2-1-01 ソースコードと実行コード

開発ツールとは

ビルドするためのツールを開発ツールと呼びます。コンパイラは最もその中心になるものでしょう。また、絵を描いたり音を取り扱うためのツールも必要です。みなさんが日頃何気なく使っているソフトウェアでも十分開発ツールになりうるわけです。

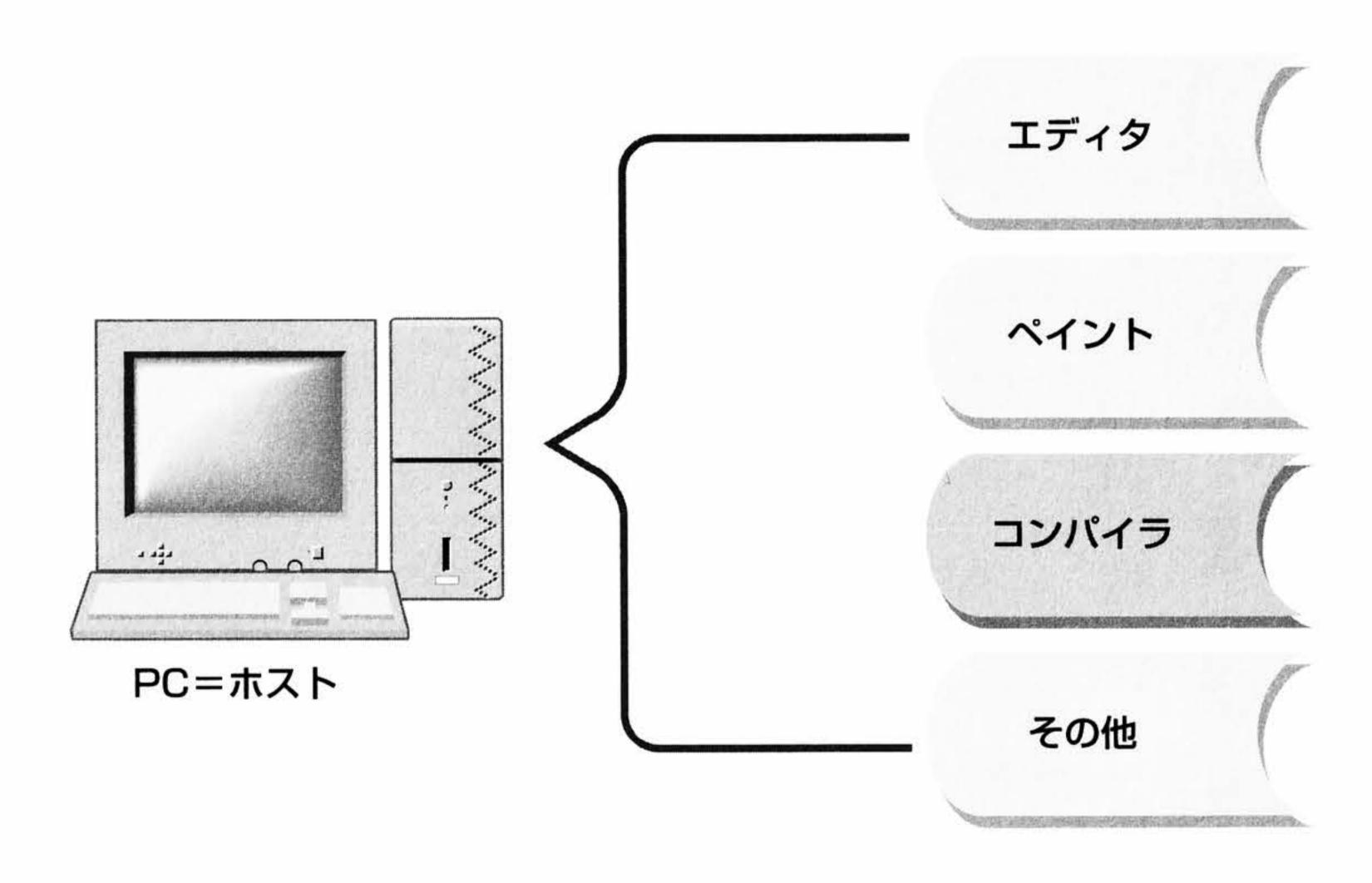



Fig 2-2-2-01 開発ツール

開発ツール紹介

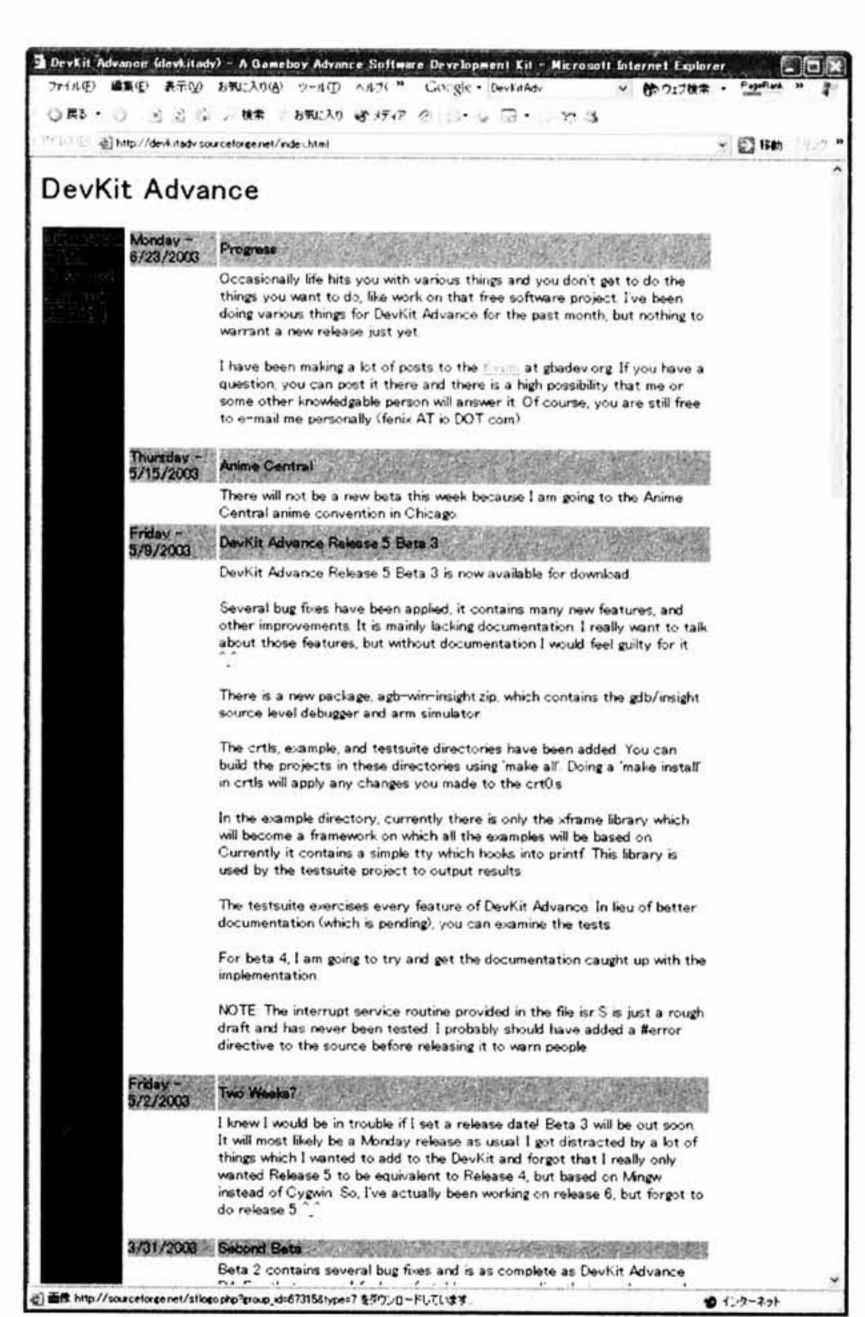
開発の中心となるコンパイラから解説しましょう。GBA以前のGB環境ではGBDKが有名でした。事実上、これがGB開発の標準といってもいい状況でした。GBAではここまで標準化されたものはありません。現在、主として2種類のコンパイラが使われています。基本的に各コンパイラはGCC (GNU COMPILER COLLECTION) がベースになっています。それぞれに特徴があるので選択に困ります。自分の好みで決めていいと思います。

GBACC

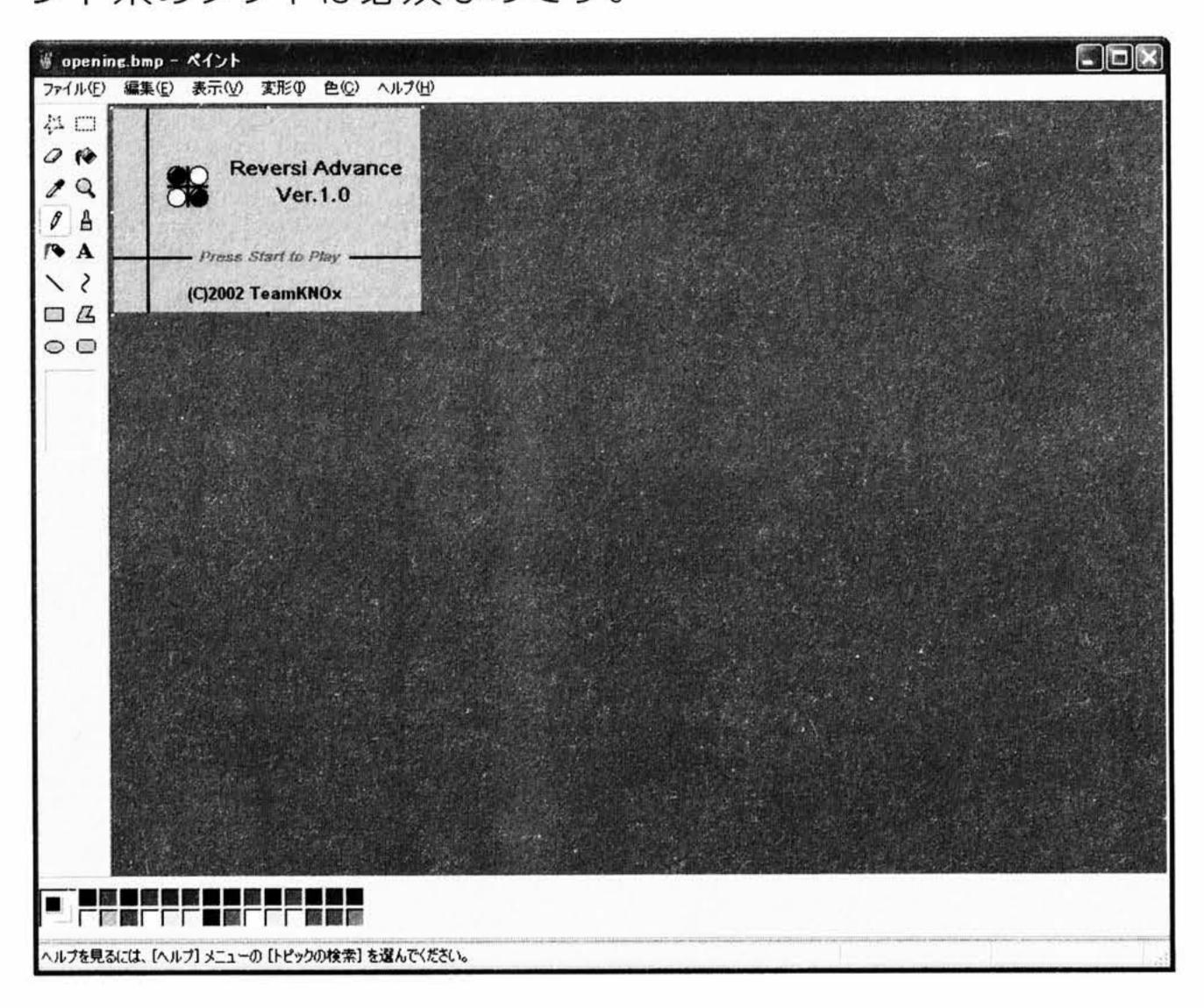
最近出てきた環境です。次のDevkitAdvがCygwin環境で動作するのに対してこちらはMinGW環境でコンパイルされているネイティブ環境です。MinGWとはMinimalistGnuWindowsの略称でWindows環境下にGnuの最低限のツールを移植したものです。ただし、DevkitAdvに比較してライブラリなどが若干貧弱ですし、設定ファイルも少し複雑です。

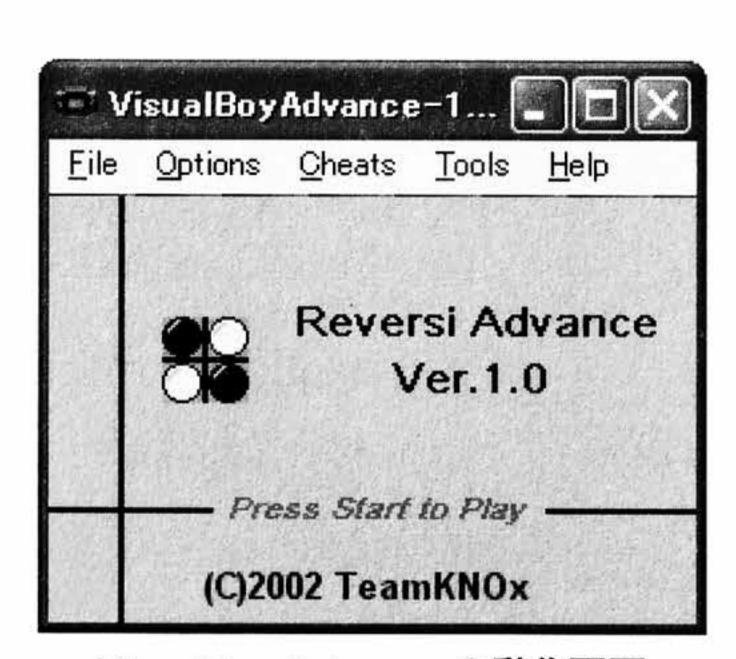
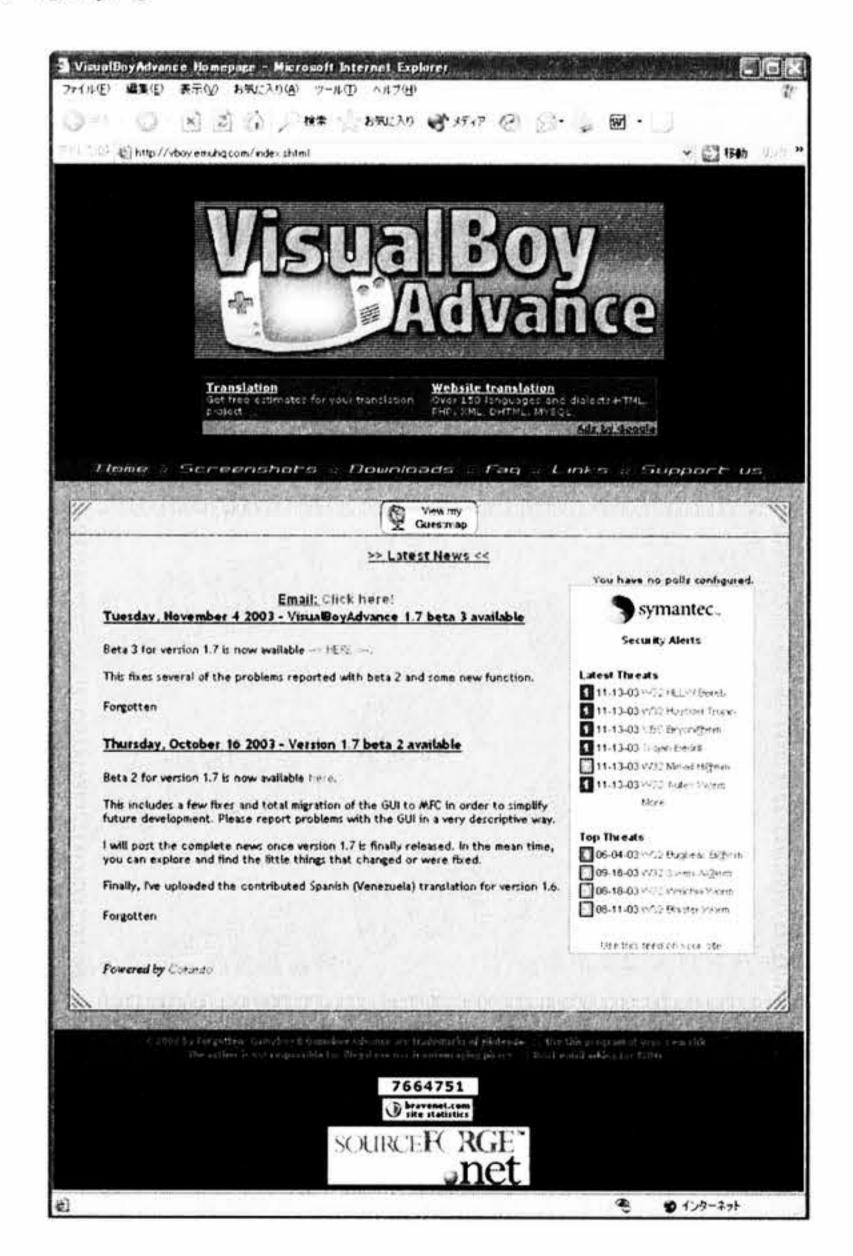
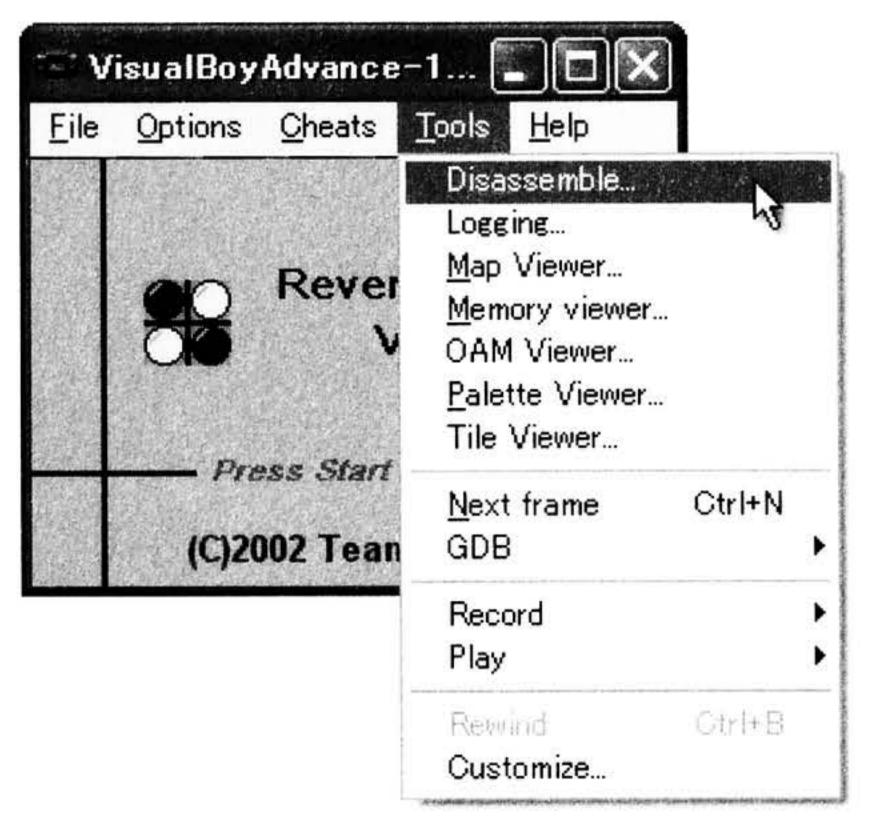
ただ、Windowsネイティブで動作するため、動作が軽いので、筆者はこちらで開発するようにしています。現在、Webが閉じられているのでダウンロードできませんが、筆者はGBACCの作者とコンタクトを取っていて、GBACCを書籍に収録する許可をもらっていました。

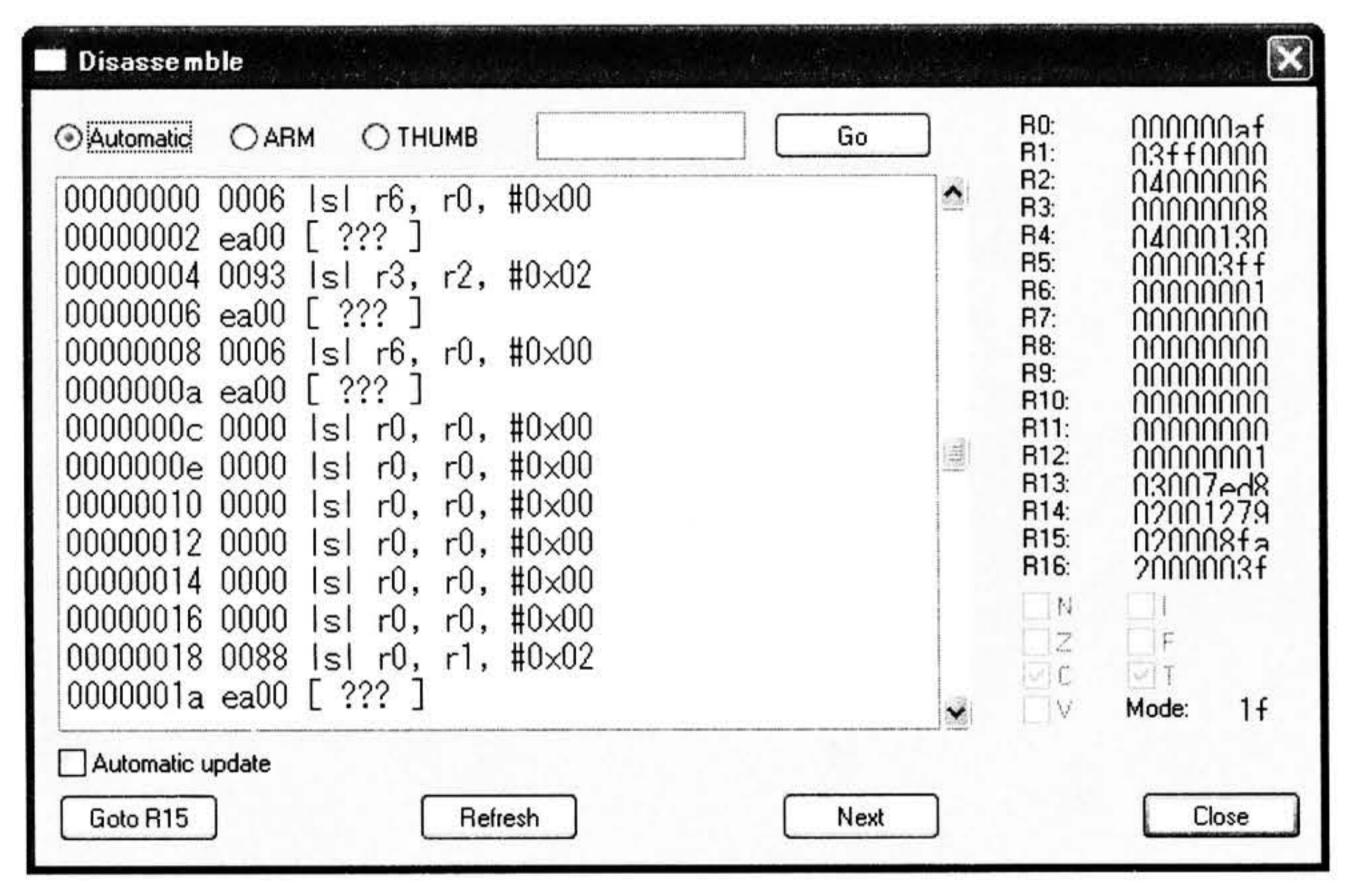
DevkitAdv

早くからリリースされたこともあり、ユーザーが多いのが特徴です。インターネット上で入手できる各種のデモなどは、これを前提にコーディングされていることが多いようです。ただ、少しずつ派生したものがあるので真の意味での標準とは言いがたいところもあります。また、動作環境としてCygwinを使わなければいけません。CygwinはUNIX互換のコマンドをDLLで呼び出しています。実行は昨今のマシンではそれほど気になりませんが、やはり、ネイティブな環境と比較するとイマイチ遅いです。このことからCygwinを使うことに難色を示すユーザーも結構いるようです（少なくとも筆者はそうです）。

DevKitAdvのホームページ
http://devkitadv.souceforge.net/index.html

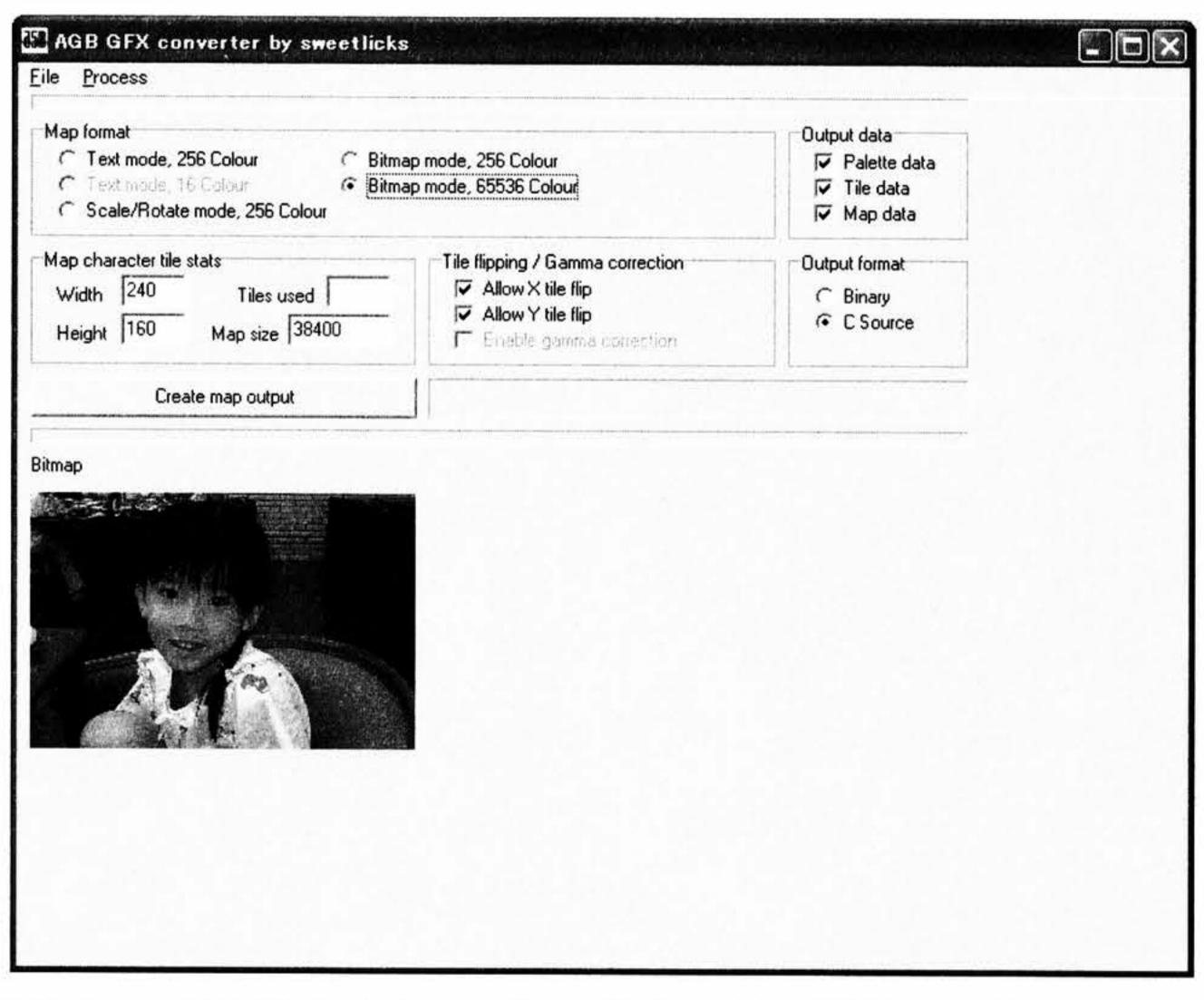
ペイントブラシ

「あれ？ これってWindowsの添付ソフトのこと？ これも使えるの？」と思った人もいるかもしれません。このソフトも十分開発環境と言えます。もちろん、もっと高機能なペイント系ソフトを持っているのならそちらでも構いませんし、もちろん使い慣れているソフトがあれば、そちらをおすすめいたします。GBAのゲームを見てもらうとわかりますが、スーパーファミコン並みのグラフィックが確保されています。GBではタイルと呼ばれるグラフィックの管理方法でしたが、GBAではビットマップファイルも扱うことができます。これらのグラフィックを作成するためにはペイント系のソフトは必須なのです。

エミュレータ

エミュレータというとダークな感を拭えない人も多いかもしれません。PC 書籍売り場に行くと、エミュレータ関連の書籍は平積み状態です。ROM を吸い出して PC で遊ぶことができるエミュレータは、それだけで魅力的ですが、開発ツールとしてみた場合も非常に重要です。基本的にはエミュレータで動作するソフトを作ることができれば、その対象（今回は GBA ですね）となるソフトを作ったのとほぼ同義になるからです。ビルドしてできた実行ファイルをドラッグ＆ドロップで即実行できるエミュレータはお手軽にテスト＆デバッグができて便利です。エミュレータもコンパイラと同様に各種インターネットで入手できますが、本書では再現性とデバッグ環境に定評がある VBA を用います。

VisualBoy Advance の動作画面

VisualBoy Advance のホームページ
http://vboy.emuhq.com/index.shtml

VisualBoy Advance の Tools メニュー

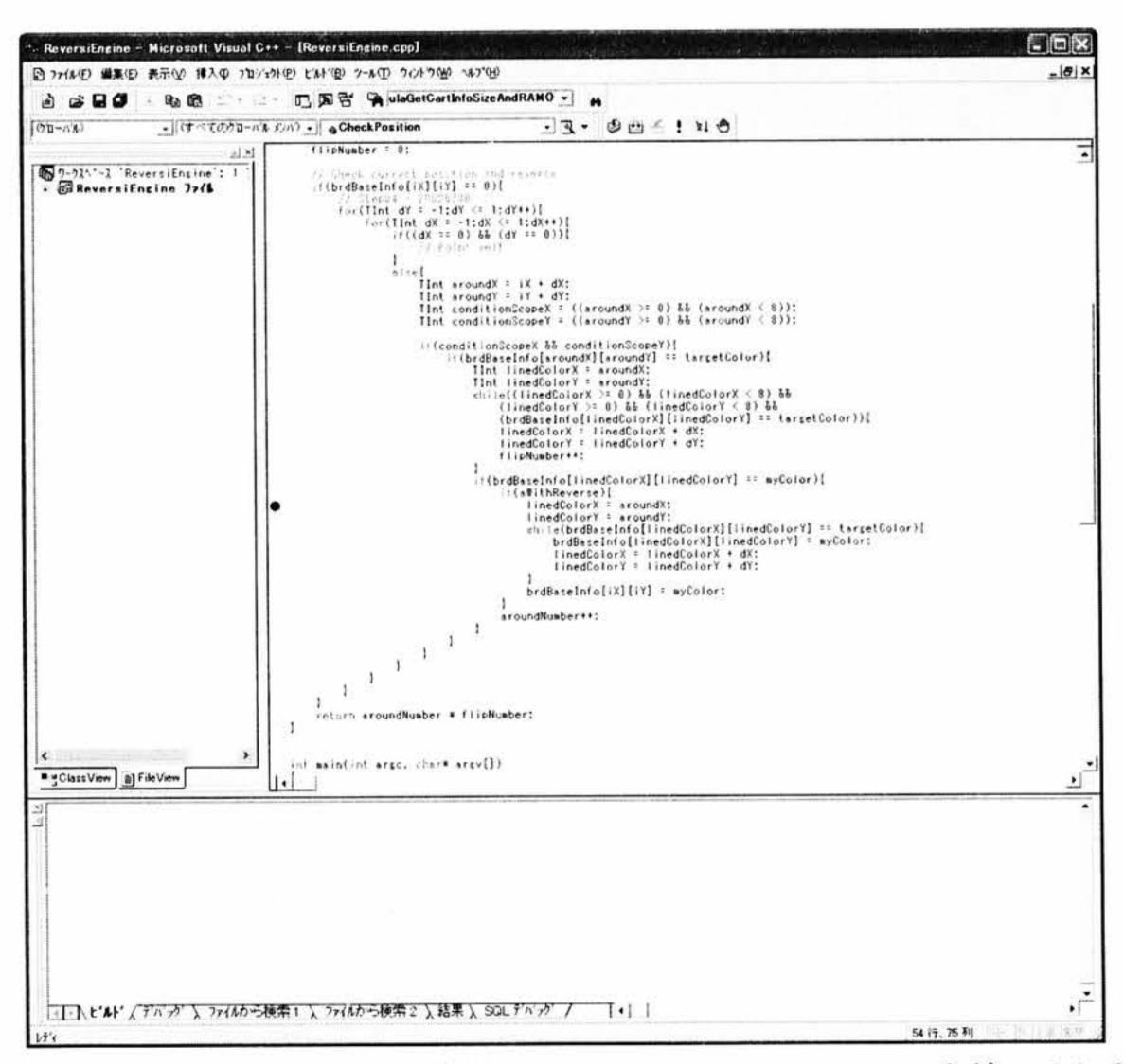
デフォルトのキーアサイン	
Up	Up Arrow
Down	Down Arrow
Left	Left Arrow
Right	Right Arrow
Button A	Z
Button B	X
Button L	A
Button R	S
Select	Space
Start	Enter
Speed	Numpad +
Capture	F12
GameStop	C

VisualBoy Advance の Tools メニューから Disassemble（デバッグツール）の画面を開いたところ。

グラフィックコンバータ

Windowsのグラフィックと GBAのグラフィックはビットマップの構成が違います。つまり、ペイントブラシで描いた絵のファイルはそのままではGBAでは使えないのです。そこでファイルを変換する必要がでてきます。このグラフィックファイルを変換するためのツールをグラフィックコンバータと呼んでいます。ペイント系のソフトによってはプラグインモジュールで直接GBAのグラフィック形式にコンバートしてくれるものもありますが、ここでは単独のソフトを用います。今回はAGB GFXを使うことにします。また、筆者らのチームで開発した圧縮フォーマットを取り扱うためのグラフィックコンバータ bmpcnv（ベタなネーミングですね）も合わせて利用します。

Windowsで作成したビットマップファイルをGBA用に変換するAGB GFX

その他のツール

今回はサウンドを使っていませんが、グラフィックと同様にサウンド関係のファイルを取り扱う場合もあります。このときは当然ながら、サウンドを取り扱う各種ソフトを使うことになります。

また、Windows用のコンパイラ（VisualC++）もゲームのエンジン部分などの記述やテストには有効です。もちろん、VisualC++はPC用なので直接GBAのプログラムをデバッグできませんが、CPUによる差がほとんどないゲームのエンジン部分などは機能の充実したVisualC++などで先にデバッグしておくわけです。きれいなグラフィックがなくても動作の過程はよくわかります。デバッグが完了したらGBAに移植します。

ReversiAdvanceのエンジン部分の開発にはVisual C++を使いました

開発ツールのインストール

開発ツールをインストールしてみましょう。この手のフリーツールの寄せ集めにはインストーラはありません。自分の手で開発ツールを的確にインストールして開発環境を構築しなければいけません。また、この開発環境の構築もソフト開発にとって非常に大事なことです。基本的には圧縮されたファイルを的確なフォルダに展開するだけなのでむずかしいことはありません。恐れずにやってみましょう。

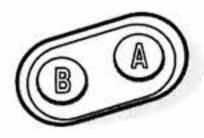
ここでは筆者の環境を例にとってお話します。自分なりのインストールポリシーがある人は、ここに書いてあることを参考に適宜読み替えてください。ただ、はじめて環境を構築するという人は、とりあえず筆者の環境と同じ環境にすることをお勧めします。開発環境について充分に理解したら、少しずつ自分に合った環境に変えていくのがいいでしょう。

筆者の開発環境（ハード）

CPU……Celeron 1.2GHz　　HDD……20GB

RAM……256MB　　OS………WindowsXP SP1

開発ツールのパス設定

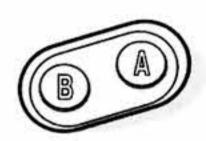
ビルドは複数のツールが連携動作して行なわれます。つまり、ビルドの最中は一連のツールが連続的に呼び出されることになります。よってパスの設定は非常に重要です。逆に言えばパス設定さえ正確に行なわれていれば、あとはほとんど問題ないと思われます。本書で使用する開発ツールでパス設定が重要になるのはGBACCだけです。bmpcnvはインストールするフォルダに注意します。VBAやAGBGFXはとくに意識しなくてもいいでしょう。

2-4-1-1 GBACC

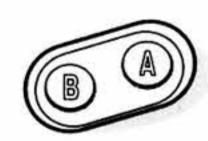
添付CD-ROMから展開するか、Webサイトからダウンロードします。筆者は**c:¥bin¥gbacc¥**に展開しました。パスは環境変数に定義してもいいのですが、一時的なパス設定をしても問題ありません。筆者は次のようなバッチファイルを作って使用するときに一時的なパス設定をしています。これは筆者のPCには複数のGCC環境が存在するためで、単純に環境変数のパス設定をしてもうまくいかないためです。GBACCを実行する前にこのバッチファイルを実行して、GBACCのコンパイラが最初に使われるようにします。バッチファイルのファイル名はわかりやすい名前で保存しておきます。ここでは、**gbastart.bat**としておきます。

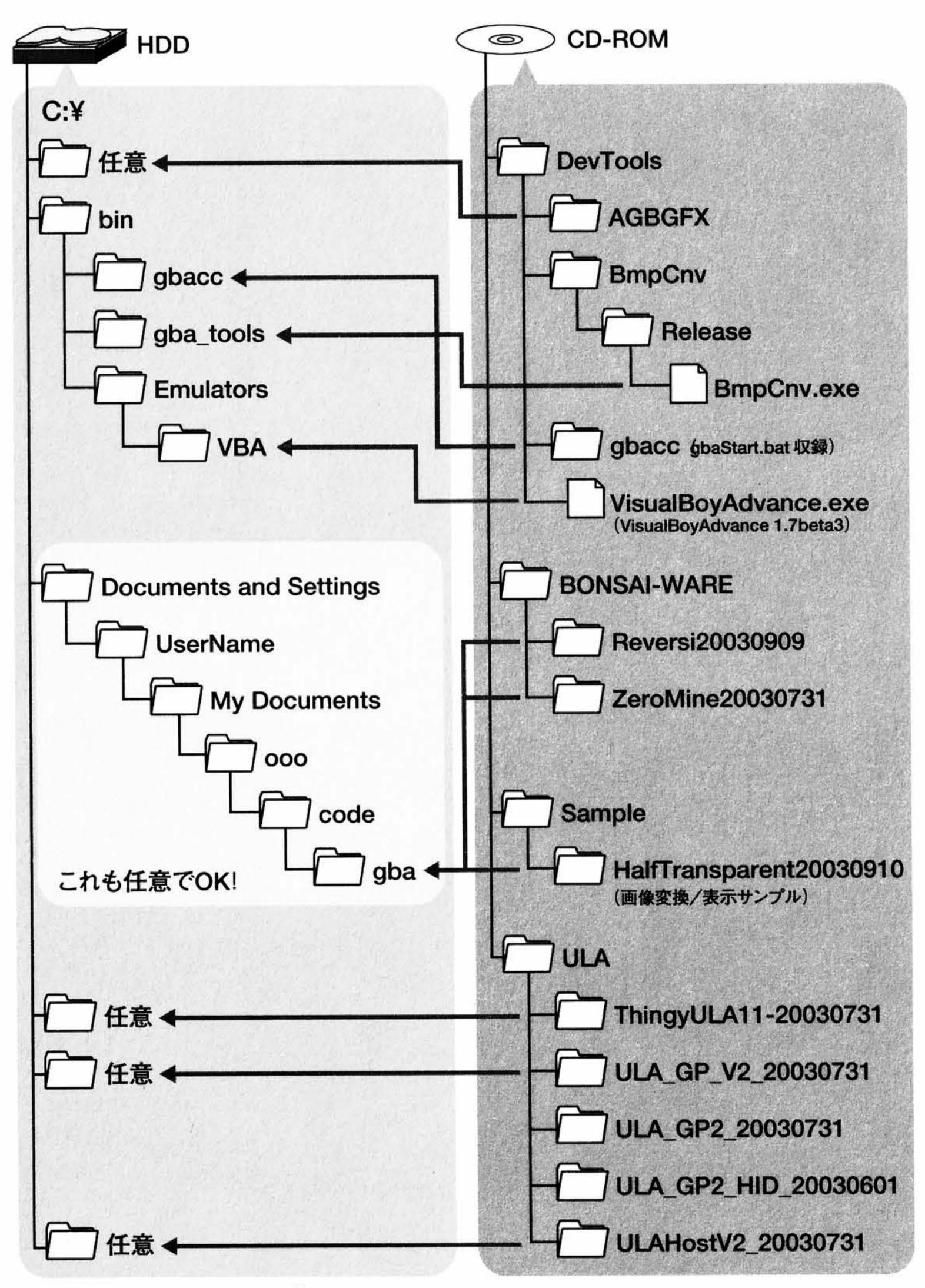
```
@echo off
set PATH=c:¥bin¥gbacc¥bin;C:¥bin¥gba_tools;%PATH%
```

● gbastart.batの内容

2-4-1-2 VBA

VBAの動作にはDirectXが必要です。筆者の環境は WindowsXPなので、とくに何の設定も必要ありませんでしたが、これ以外の環境ではDirectX Ver.8以降をインストールしておく必要があります。DirectXのインストール方法は付属マニュアルかMicrosoftのWebサイトを参照してください。VBAは**C:¥bin¥Emulators¥VBA**に展開しています。コマンドラインから実行することはないので、パスの設定は行ないません。

2-4-1-3 bmpcnv

bmpcnvは**C:¥bin¥gba_tools**に実行形式のみを展開してあります（bmpcnv.exeのみを保存します）。パス設定はgbaccのパス設定時に同時に行なわれます（gbastart.batを使用する場合）。

2-4-1-4 AGBGFX

これも通常のWindowsソフトウェアなので、とくにパスの設定は行ないません。そのつど動作させます。好きな場所に展開してください。

※付属CD-ROMから起動ディスクへのインストール(本書でのパスの設定)

※CD-ROMに収録している各ファイルはZIP圧縮してあります。ハードディスクの適当なディレクトリにコピーし、圧縮ファイルを解凍してからインストールしてください。

開発ツールの動作確認

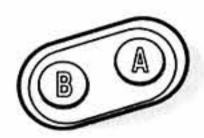
インストールが的確に行なわれたかどうか、開発ツールの動作確認を行なってみましょう。ここでうまく動作しないとビルドできません。また、動作確認の方法を理解しておくと、開発する環境が変わった場合などにも応用が利くようになります。

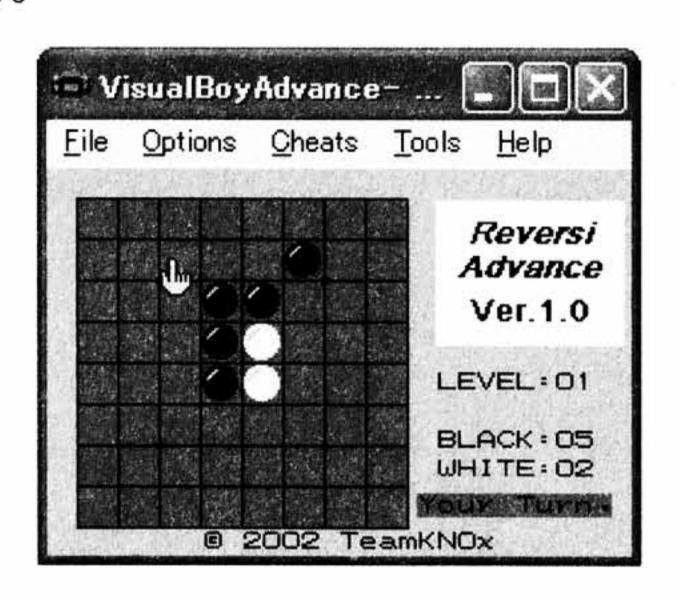
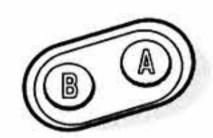
2-4-2-1 GBACCとbmpcnvの動作確認

GBACCとbmpcnvの動作確認は、当チーム作のReversiAdvanceをビルドできるかどうかで判断することができます。このビルド作業にはGBACCとbmpcnvの両ツールが必須だからです。ReversiAdvanceはTeamKNOxのWebからダウンロードするか、添付のCD-ROMからコピーして準備しておいてください。ちなみに筆者はWindowsでのファイル管理を**C:¥Documents and Settings¥UserName¥My Documents¥ooo¥code¥gba¥(ProgramName + yyyymmdd)**のようなディレクトリで行なっています。以下にGBACCを実行させてGBAのソフトを作るための手順を列記します。

1. コマンドプロンプトを立ち上げる
2. gbaStart.batを実行する(ドラッグ&ドロップでよい)
3. プログラムが格納されたフォルダに移動する
4. make1.batを実行する

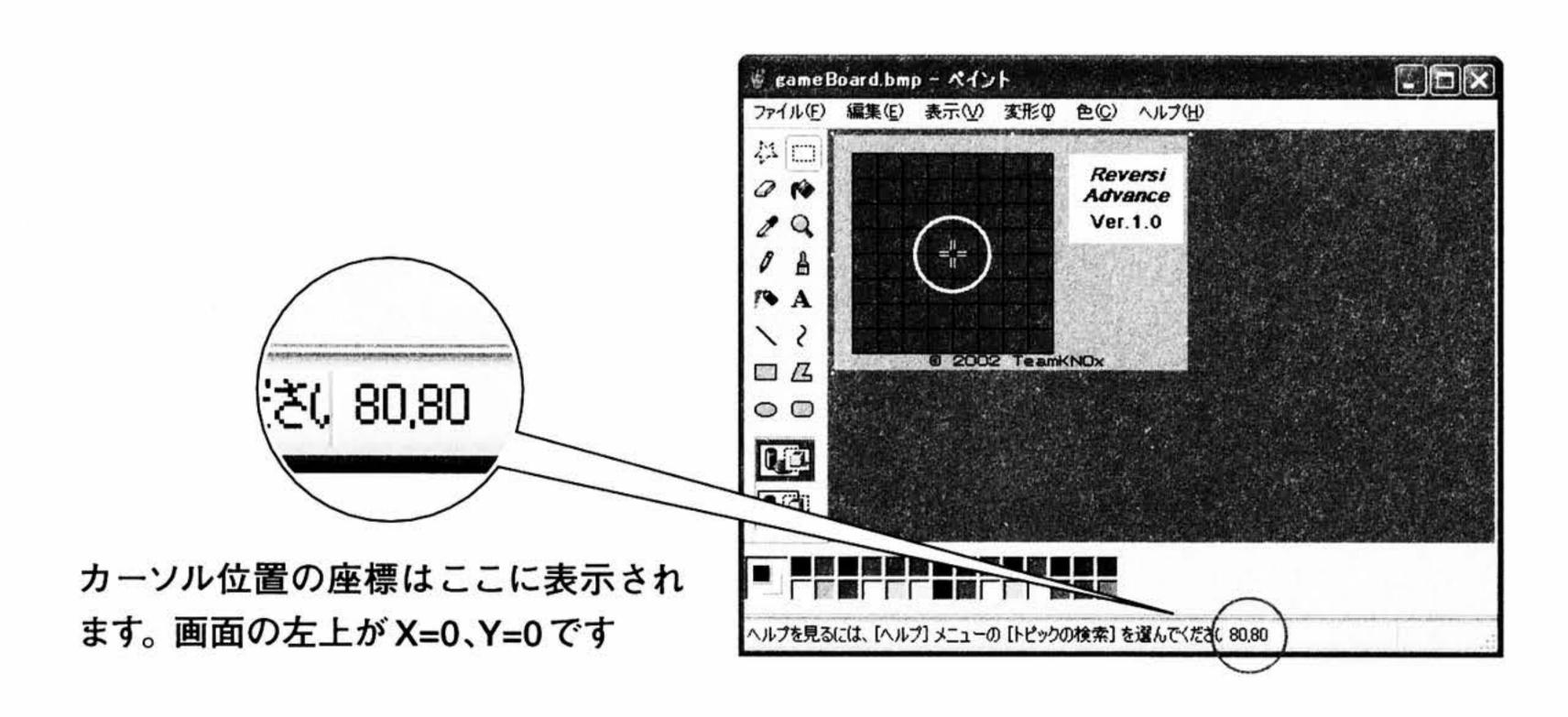
コマンドプロンプトを起動したら(スタートメニューのアクセサリにあります)バッチファイルgbaStart.batをコマンドプロンプトのウィンドウにドラッグ&ドロップします。すると、C:¥bin¥gbacc¥gbaStart.batと入力されるので、ここでEnterキーを押すとgbaStart.batが実行されます。次にcd(半角スペース)と入力してReversi2003909フォルダをコマンドプロンプトのウィンドウにドラッグ&ドロップします。すると、そのフォルダまでのパスが入力されるので(ここでは、"C:¥Documents and Settings¥Osamu OHASHI¥My Documents¥ooo¥code¥gba¥Reversi2003909"になっています)Enterキーを押します。続いてmake1.batと入力してEnterキーを押します。

ソースを何も変更していなければ、ReversiAdv.mb.gba(Reversi AdvanceのROMイメージです)ができあがります。これがエミュレータで問題なく動作すれば、GBACCとbmpcnvの動作確認ができるわけです。

2-4-2-2 VBAの動作確認

エミュレータが正しく動作しているかどうかは、ターゲットマシンのROMイメージを実行させて確認します。添付CD-ROMに同梱されているReversiAdvance(ReversiAdv.mb.gba)を実行させます。正しく動作するようであればVBAの問題はありません。

2-4-2-3 ペイントの動作確認？

このソフトは、Windowsをインストールすると標準でインストールされるツールです。メニューから選ぶだけで起動します。少しお絵かきなどをしてみるといいでしょう。

カーソル位置の座標はここに表示されます。画面の左上がX=0、Y=0です

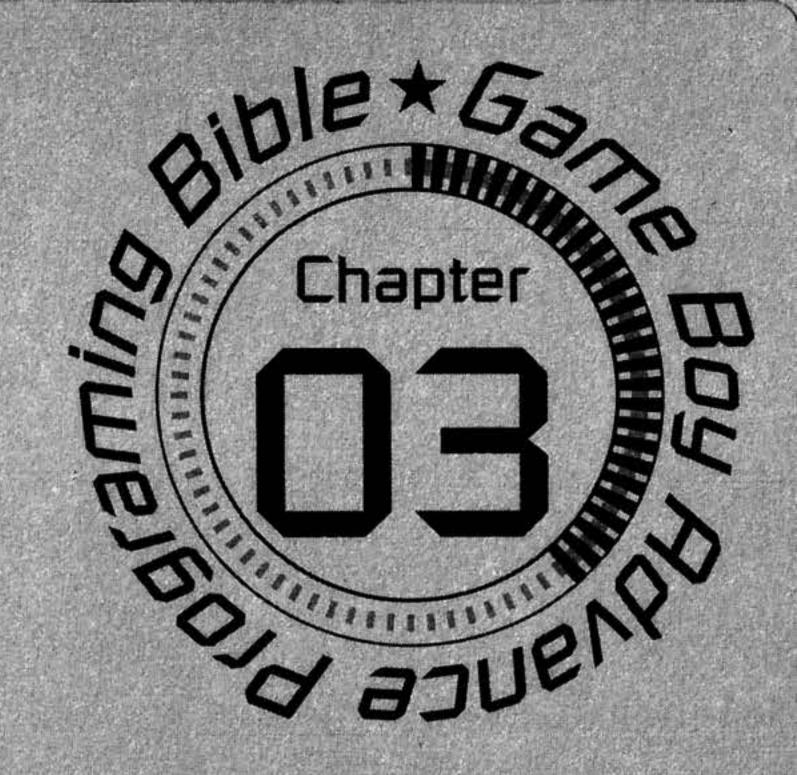
Column

クロス開発秘話

いまでこそクロス開発ではPCを使うのが一般的になっていますが、筆者が駆け出しのエンジニアの頃は専用の開発ツールで開発を行なっていました。お世辞にも使いやすいとは言えないツールで、ソースコードを入力し、ビルドして実行コードを得て、ターゲットに実行させます。基本的な開発サイクルはいまのシステムとほとんど変わりませんが、現在のものと比較すると随所に使いにくさの罠が仕掛けてありました(今日のシステムが、過去の反省を踏まえて改良されてきたと考えるほうがいいかもしれませんね…)。それでも、やっていることは昔とそうは変わりません…。これは、幸せなのかそれとも…?

Column

GCCとCygwin

フリーの開発環境ではGCCは外せない存在です。GCCは元々GNU C Compiler(グニュ・シー・コンパイラ)の略称でしたが、その後さまざまなコンパイラのリリースが行なわれたことから、CではなくGNU Compiler Collection(グニュ・コンパイラ・コレクション)と改称されました。

もう一方のCygwinですが、こちらは元々GNU関連の各種ツールのサポートなどを行なう企業CygnusSolution(キグナスソリューション)がリリースしていたソフトウェアです。同社はのちにLinuxの最大手ディストリビュータであるRedhatに買収されて今日に至っています。

Game Boy Advance プログラミングⅠ

ゲーム制作でGBAのプログラミングを学ぶ

GBAで動作するプログラムとして作成するゲームの内容と、プログラムを組み立てるために必要となるハードウェアやデータ形式について考えます。また、あわせてWindowsのビットマップデータをGBA用に変換するAGBGFXの使い方も説明します。

GBAでゲームを作る必然性

筆者はTeamKNOxというプライベートのシステム開発集団に所属しています。このチームはGBCでの開発活動が長いこともあり、GBCで筆者らが面白いと思う大抵のことはやってしまいました。GBAのプログラミングを行なうにあたり、GBAでなければできないことをやりたいと思います（そうでなければGBCでやればいいのです）。最近になっていくつか候補が見つかりました。

① ARMプログラミング環境の理解

② GBCでは実現できないくらい大規模なもの

今回はプログラミングということで、②の「GBCでは実現できないくらい大規模なもの」を中心に考えていきたいと思います。①の「ARMプログラミング環境の理解」は、開発環境の整備やブートストラップコードを読むこと、今回のプログラムづくりで、ある程度カバーできるのではないかと思っています。

大規模なソフト??

GBCに比べて強力なグラフィックやサウンドと、それを処理する処理部の大規模なものを作ればGBAで作る名目は立ちます。ただ、一体誰がこの「大規模なデータや処理」を作成するのでしょうか？

商用ゲームのように仕事としてなら作れるかもしれません。しかし、プライベートで開発する場合に、この論理は成り立ちません。むしろ大規模なデータよりも、GBAのGBCに比べて強力な処理能力を前面に押し出したソフトということになるでしょう。CPUパワーを必要とするソフトの代表的なものに人工知能系ソフトがあります。また、せっかく時間をかけて作るのですから、作ったあとも末永く楽しめるソフトにしてみましょう。

CPUパワーを使うソフト?

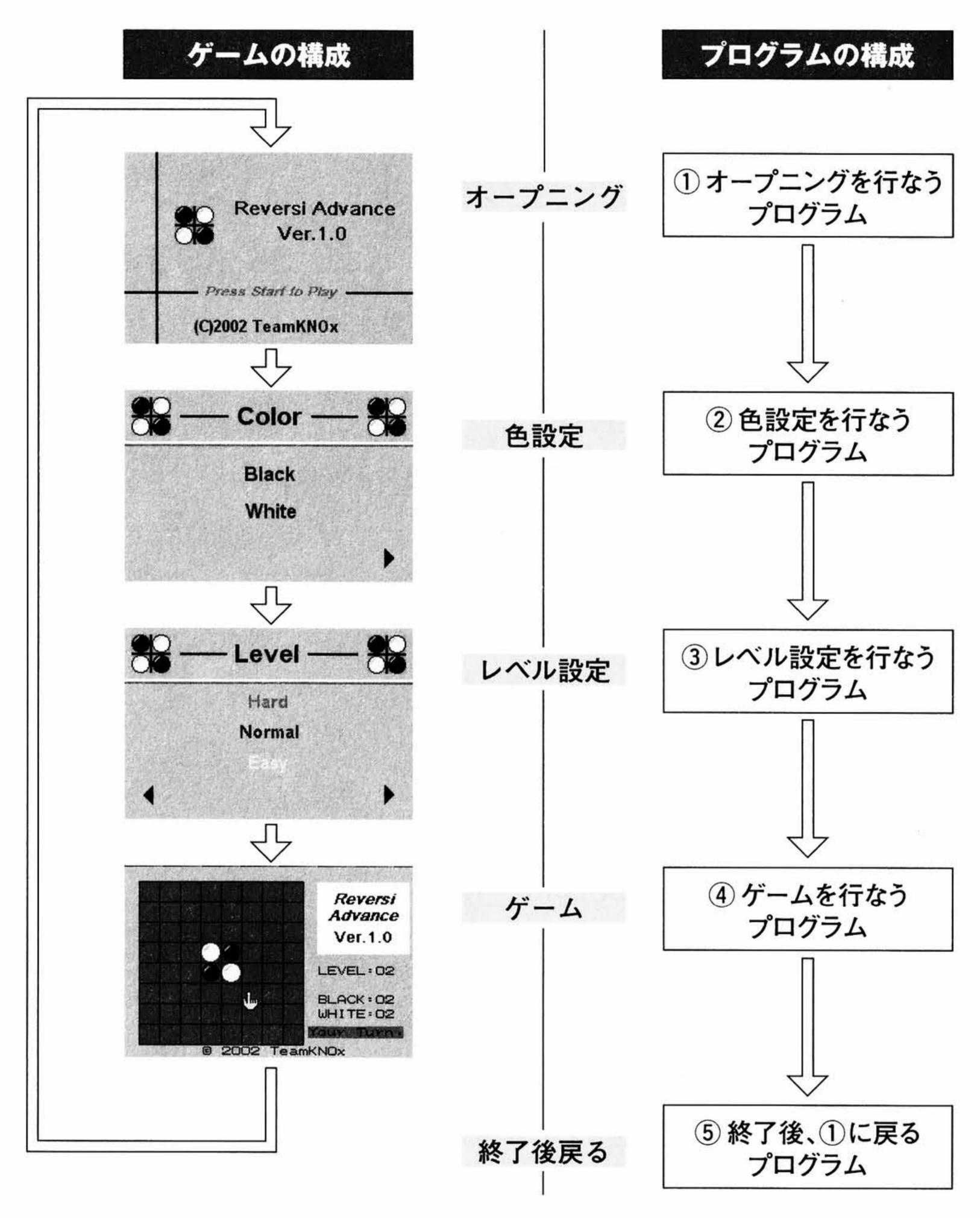
人工知能系のソフトの代表格といえば、ふだん遊んでいるボードゲームがそうです。とくに将棋や囲碁、チェス、リバーシ（オセロゲーム）はPCやゲーム機などでも数多く出ていますし、老若男女問わず楽しめるゲームだと言えるでしょう。ただ、チェスなどは既にコンピュータが人間の能力を完全に凌駕してしまっています。しかし、リバーシは処理能力の高い最近のPCでは人間を凌駕していますが、GBAのように限られたリソース（CPUパワー、メモリ、そして…開発時間？）では、その状況は一変します。

リバーシ(オセロ)ゲームの制作(BONSAI－WARE)

以上のことから、リバーシをここで制作することにします。基本的には、「省メモリ」「実行速度」「そこそこの強さ」を目標に作ることにします。

省メモリには、大きなプログラムの開発では開発工程も長く、途中で飽きてしまうことを避ける意味もあります。実行速度も重要です。とくにこの例のようにコンピュータと対戦するタイプのゲームは、言い換えるとコンピュータに待たされるタイプのゲームです。あまり、長い時間待たされるとゲームの途中でイライラしてつまらなくなってしまいます。

ゲームの構成

リバーシは、相手と自分の持ち石を黒か白と決めて交互に石を置いていき、相手の持ち石の並びを自分の持ち石で挟むと、その持ち石を自分の持ち石にできます。最後に多くの持ち石を盤面に置いてあるほうが勝ちとなるゲームです。ここからゲームの構成を考えてみます。まず「色の設定」が必要です。次に上記の「実行速度」は「コンピュータの強さ」とのトレードオフになるので、「コンピュータの強さ」を決めるレベル設定も必要です。さらにゲーム(プログラム)がはじまって、いきなり「レベル設定」「色設定」を行なうのも何ですから、タイトル表示もほしいところです。まとめると、「オープニング」「色設定」「レベル設定」が最低限必要となるわけですね。

上左のようにゲームの構成を考えると、プログラムの構成は上右のように、これに準じたものになります。なんか、ふざけているようですけど、大真面目です。つまり、①〜⑤に相当する単独のプログラムを作成して、つなぎ合わせれば、ひとつのゲームシステムとして動作するわけです。

ゲームの構成要素の分析

ここまでは見た目(画面)単位での構成の分類でした。次は仕事単位でのプログラム構成の分類を行なうことにしましょう。各画面にもよりますが、以下のような要素に分類することが可能です。

① 文字を出す
② 画面を出す
③ ゲームパッドの読み取り
④ カーソルの移動
⑤ 画面の切り替え(場面設定)

それぞれの動作を細かく見ていくことにしましょう。また、必要に応じてサンプルソフトも作っていきましょう。

文字を出す

プログラムを学ぶときの最初のサンプルは、画面に "Hello World !!" を出力することです。これがいつから行なわれているかは定かではありませんが、カーニハン&リッチーの「プログラミング言語C」あたりからだと思います。このなかで、標準出力(CRT)に文字を出す最初のサンプルが

"Hello World !!" だったように筆者は記憶しています。それがポピュラーになり、さまざまな言語の最初のサンプルとして "Hello World !!" が使われだしたようです。通常、C 言語で文字を出力するには printf 関数を用いますが、ここでは同等の働きをする DrawText()を用いて出力します。

PC でのプログラミング例：

```
int main()
{
   printf("Hello World¥n");


   return 0;
}
```

GBA でのプログラミング例：

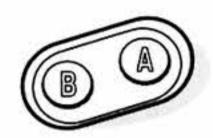
```
int AgbMain()
{
   DrawText(10, 10, "Hello World !!", );
}
```

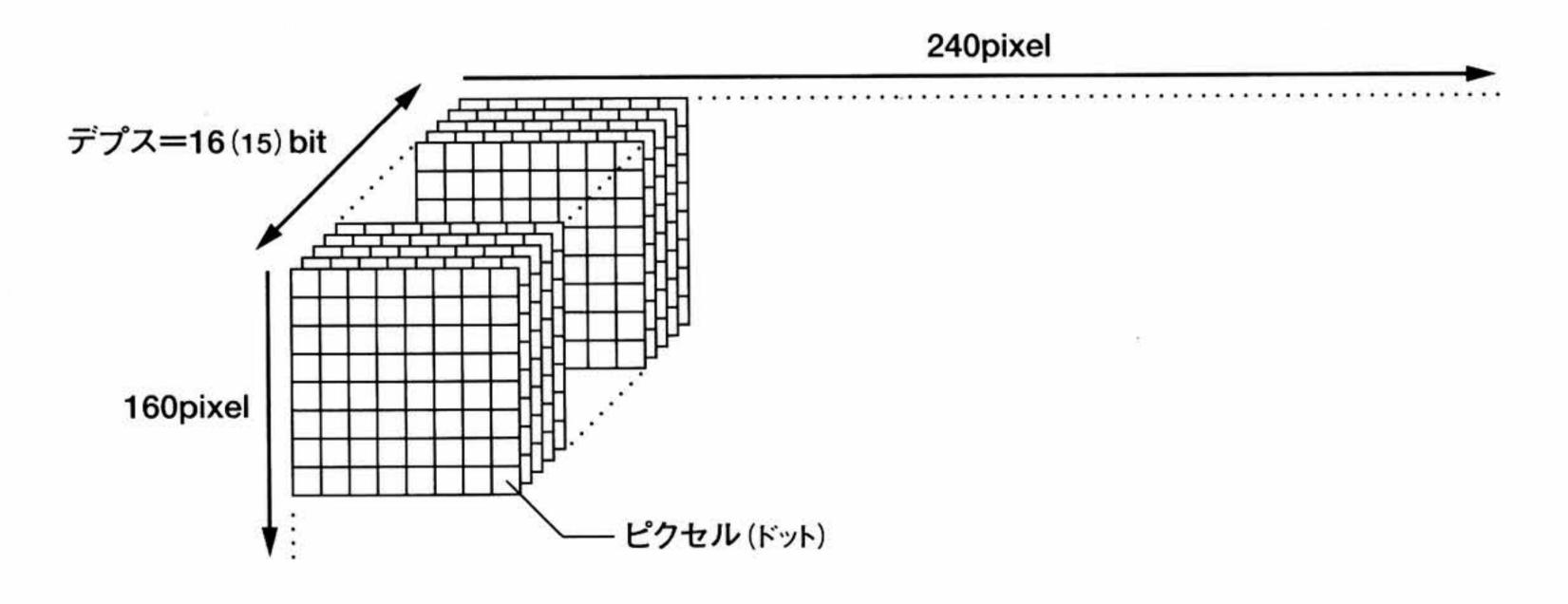
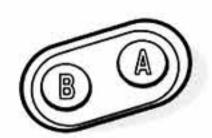
GBAに文字列を表示したい場合は、printf関数では実現できません。ここで使用しているDrawText関数はGBA用のオリジナル関数でTeamKNOxLibに収録しています。

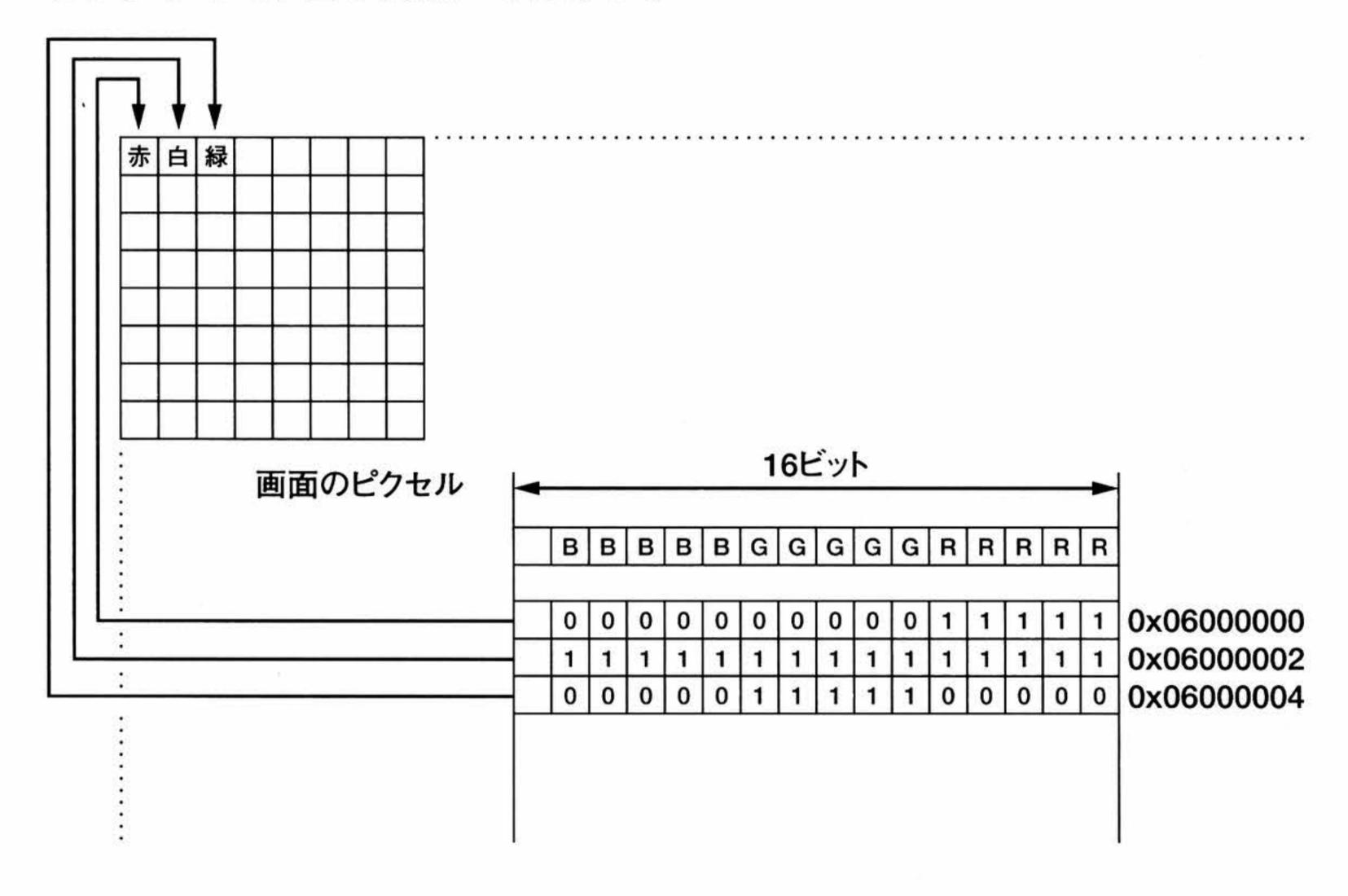
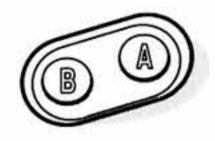
いろいろな関数が書かれていますが、これは GBA で画面表示するために必要な処理です。PC などの標準出力（コンソール）では文字を基準にして 80x25 などと画面構成が決まっていますが、GBA では画面出力はテキストではなくビットマップで行なうことができるので、ソフトの組み方にもよりますが、240x160 のピクセル単位で画面の表示する場所を決めることができます。そのため、DrawText では最初のパラメータとして文字を表示する場所（座標）を与えています。同様に PC でも Windows などの GUI 環境では座標をピクセル単位で指定して文字を表示します。

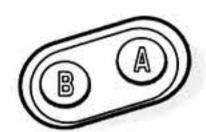
画面に絵を出す

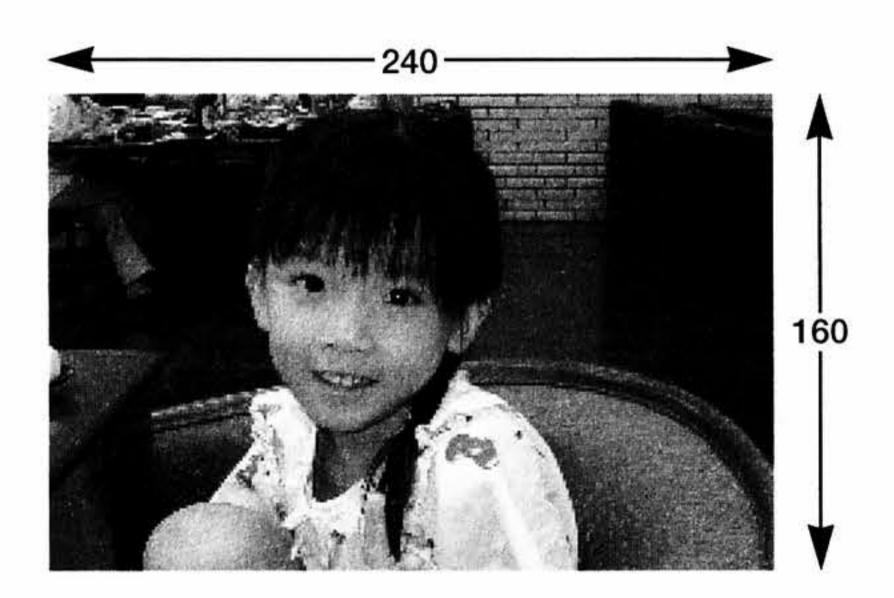
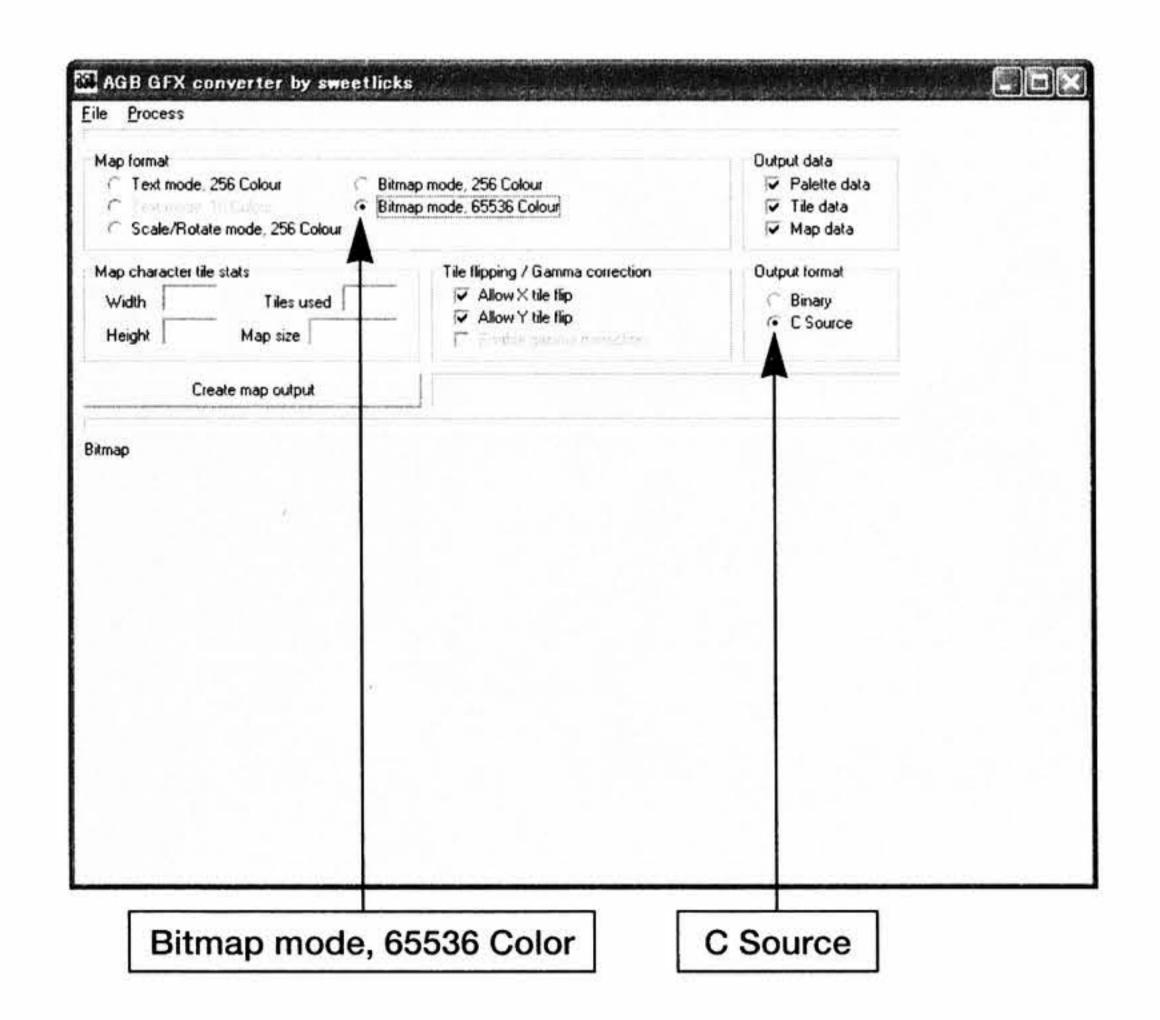
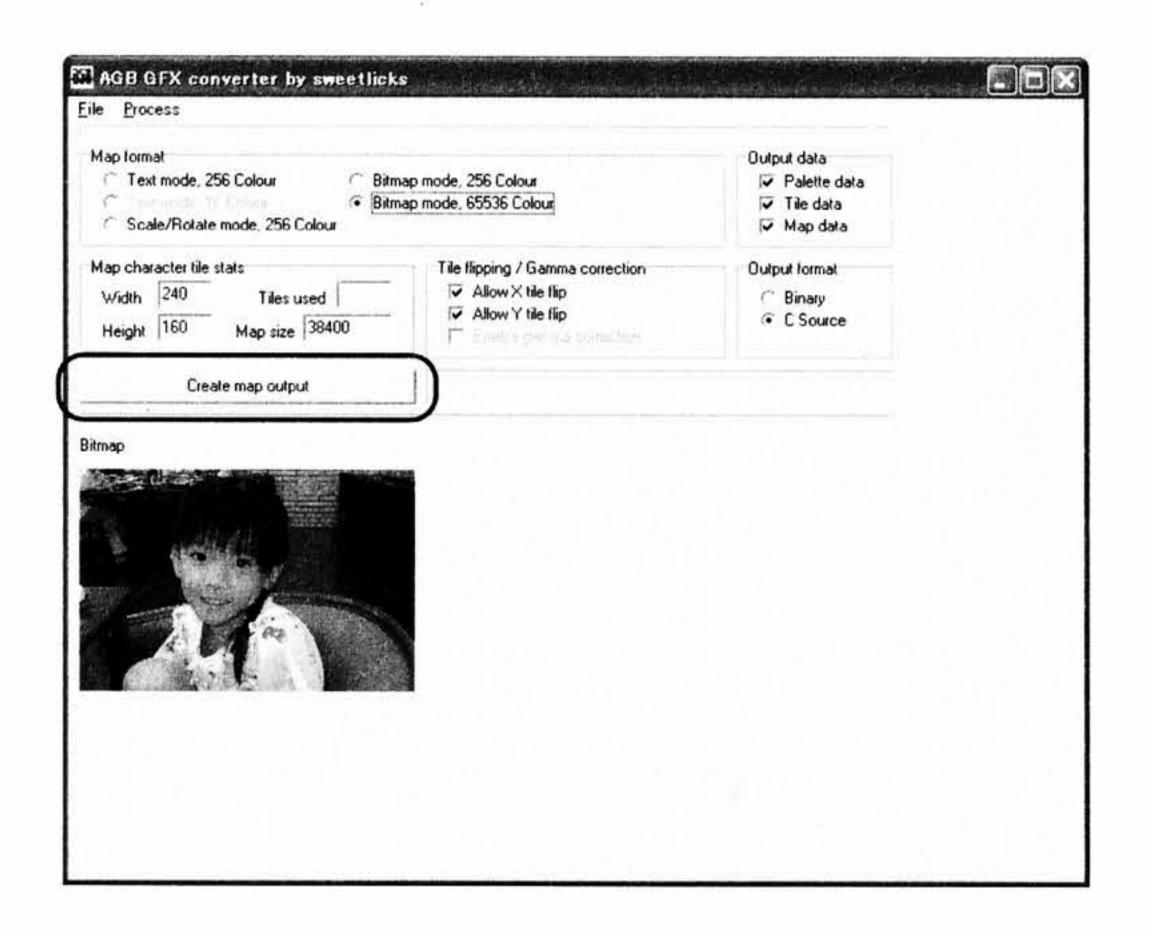
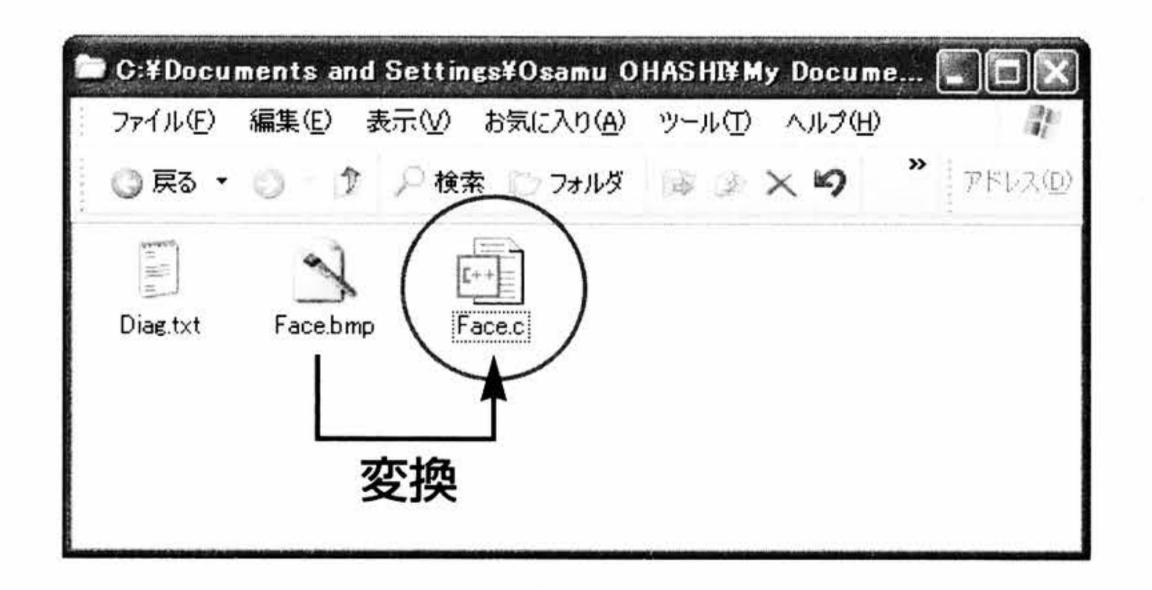
プログラミングにおいてユーザーへの訴求力が最も高いのが、画面に何かを表示することです。ディスプレイを含めて何らかの出力デバイスがコンピュータシステムにはたいてい付いています。これがないと、何か処理をしてもユーザーに知らせるすべがありません。

コンピュータはアドレスに配置された書き込み可能メモリにデータを格納することができます。このアドレス空間（なぜ、空間という言葉を使うかは不明です。慣習的にこう呼びます）にデータを書いたり、あるいは読み出したりして処理を進めていきます。画面についても同様です。

3-4-2-1 VRAM

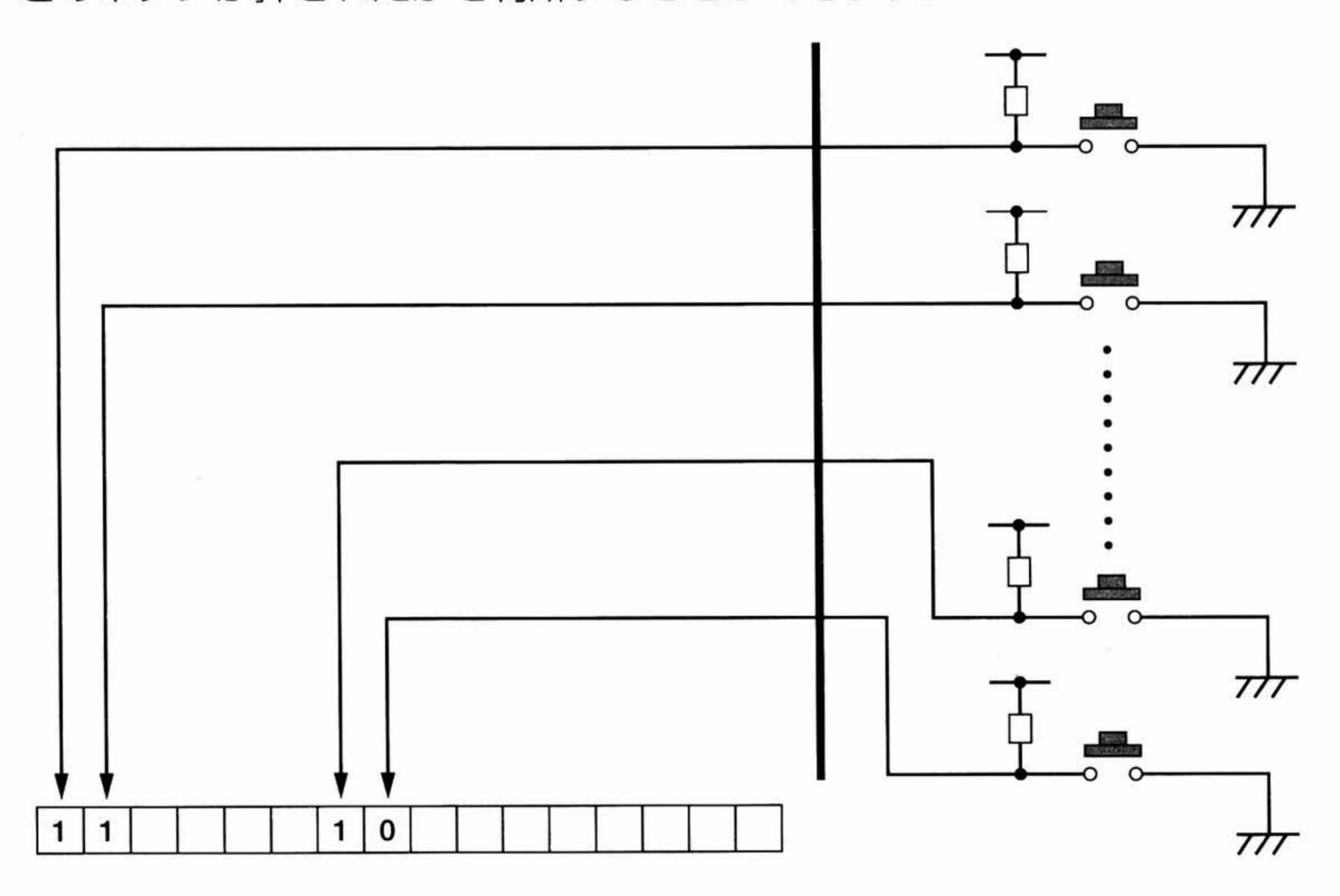
画面のピクセル（ドットとも言います）に対応しているアドレス空間があります。ここに特定の値を書き込むことによってピクセルに色が付くわけです。この特定の空間はとくにVRAM（VideoRAM）と呼ばれています。ちなみにビデオカードのVRAMの容量とはグラフィックで使えるメモリの容量を示すわけですね。話を元に戻すと、このVRAM領域に何らかの値を書き込めばいいわけです。

3-4-2-2 VRAMへの書き込みと画面出力

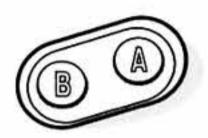
GBAにも当然VRAMがあります。モード3というグラフィックモードでGBAはフルビットマップを扱えます。これには0x06000000のアドレスが割り当てられており、図のようにこのアドレスに特定の値を書き込めば画面に色の付いた点が表示されます(0を書き込めば黒、0x7FFFを書き込めば白など)。実際には画面モードや画面のオン／オフの制御などもありますので、若干複雑になります。

3-4-2-3　1画面分の表示

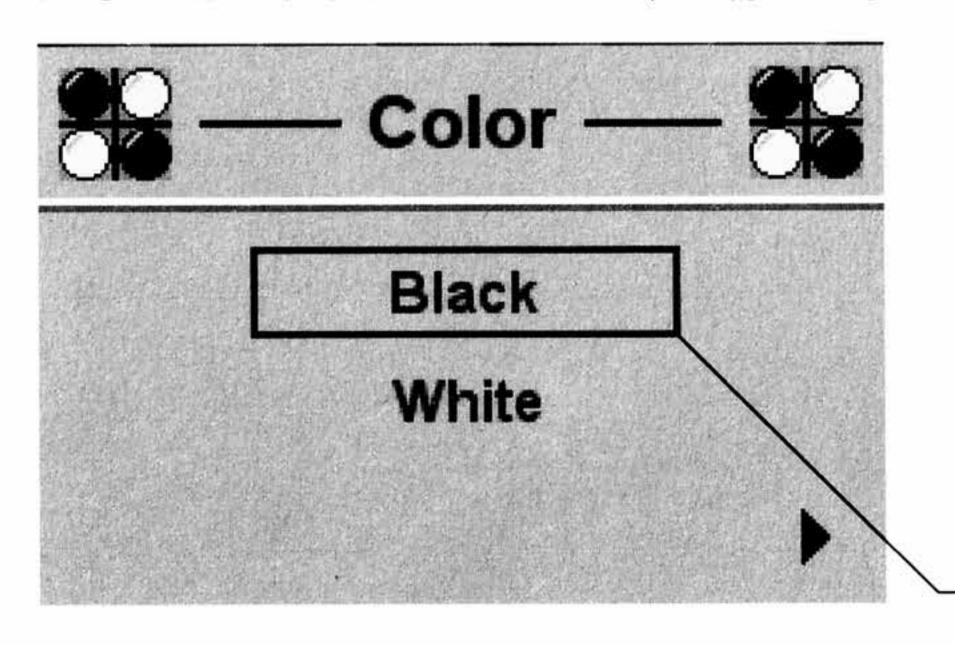
1ピクセルを地道に240×160くり返せば1枚の画面ができ上がります。画面データをあらかじめ保持しておいてそれを転送するわけですね。データの作成は先に紹介したグラフィックコンバータAGBGFXで行ないます。

3-4-2-4 AGBGFXの使い方

①まず、240x160 の24bitのビットマップデータを用意します(サンプルを使ってもOKです)。

② AGBFXを起動し、図のように設定します。ここではMap Formatを65536色のビットマップモードで、Output formatをCソースにします。Output dataとTile flippingはすべてチェックしておきます。

③①で用意したデータを開き、[Create map output] ボタンをクリックします。

④ビットマップ形式のファイルからCのソースコードが出力されます。コンパイル時にこのソース（グラフィックデータを配列に格納したもの）を同時にリンクしてグラフィックデータをプログラムに保持します。

ゲームパッドの読み取り

いままでは画面への表示など出力に関するものでしたが、こんどは入力について考えてみましょう。コンピュータに動作を指令するには、入力するためのデバイスが必要です。マウスやキーボードがこれにあたります。GBAにはどちらもありませんが、ボタンなどがこれらの代用になります。

このボタンの読み取りについて簡単に解説しましょう。ボタンをモデル化するのは非常に簡単で、下図のようになります。数が増えてくるとマトリックスにする場合もありますが、ゲームのように複数入力が必要なケースでは、同時にボタンが押された場合にわからないので、入力１つに付き入力ポートを１つ割り当てるのがいいでしょう。GBAの場合はボタン１つに付き入力１つが割り当てられています。また、ボタンの１つ１つがつながっているアドレスがあります。そのアドレスの値を見ることにより、どのボタンが押されたかを判断することができます。

3-4-4 カーソルの移動

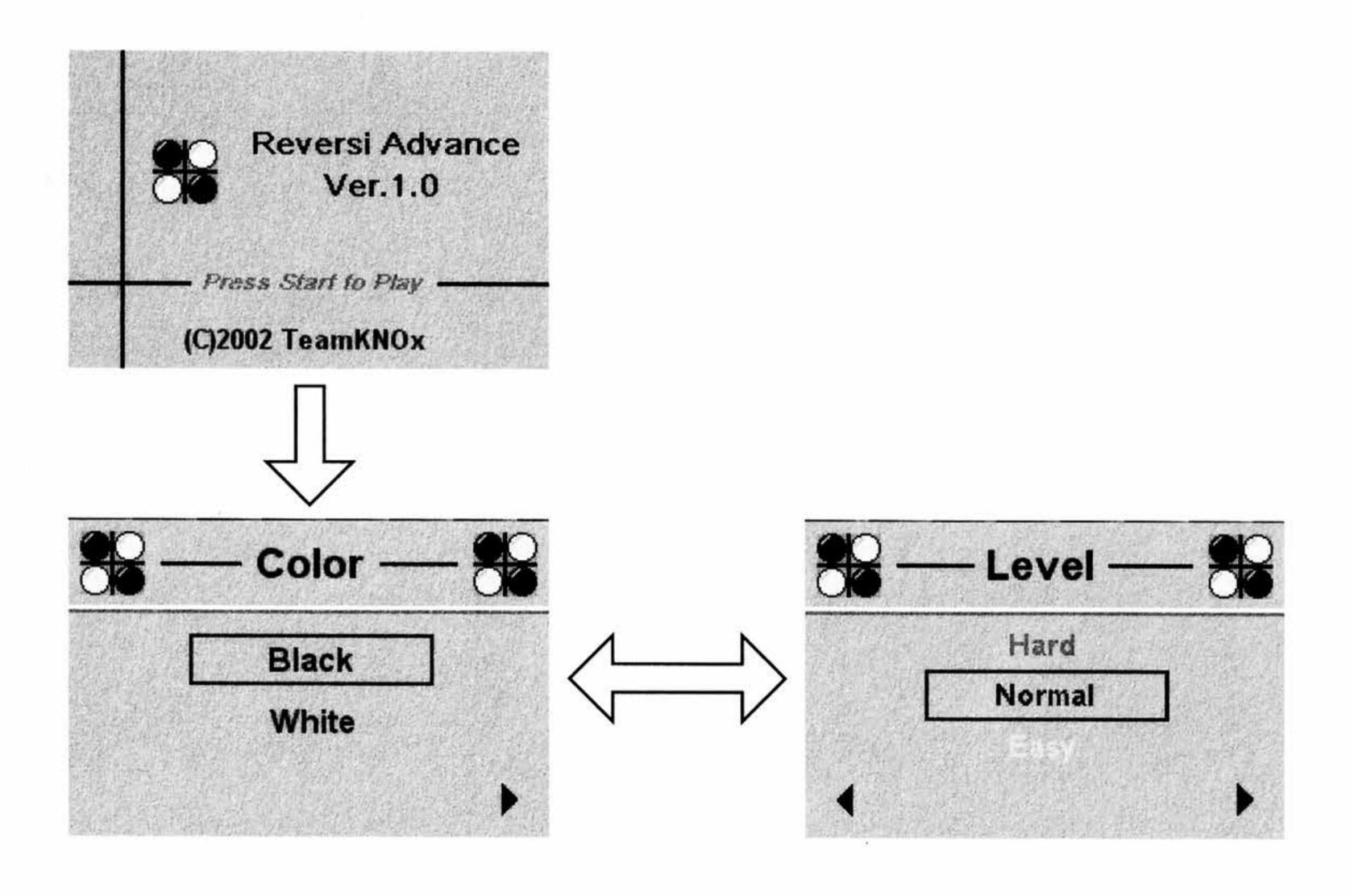
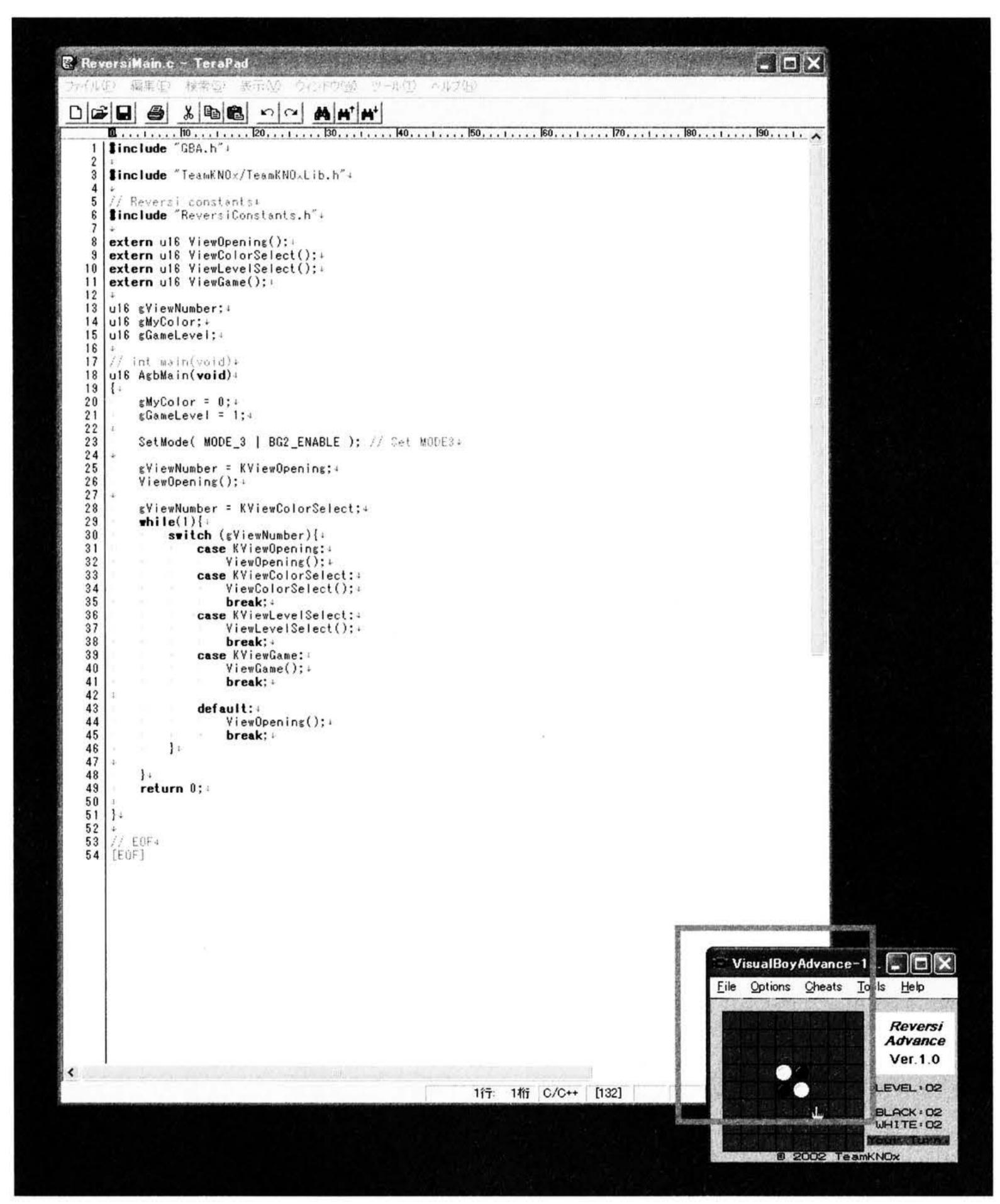
カーソルの移動は入力と出力の組み合わせです。ボタンの入力にしたがい、カーソルがそれにあわせて表示されます。選択は十字ボタンの上下でカーソルを移動して、決定は決定用のボタンを入力することで行ないます。

3-4-4-1 カーソルの形状

次にカーソルの形状を見てみましょう。カーソルは四角い枠のかたちにになっています。つまり、４本の線で選択の対象物を囲います。

画面はペイントソフトを使ってビットマップ画像として作成しますが、カーソルはプログラムで描きます。そのとき必要となるカーソルの座標やサイズなどは、ペイントソフト上で座標やサイズを測定し、メモしておきます。

Column **チェスとリバーシ**

この２つのゲームは、コンピュータが世界チャンピオンになっていますが、人間同士／コンピュータ同士でゲームさせると強弱がはっきりするので面白いでしょう。チェスの世界最強プログラムは特別なハード（ディープブルー by IBM）が必要なので実行はむずかしいのですが、リバーシは最高峰プログラム（ゼブラ）が比較的容易に入手できるので試してみるといいでしょう（最強プログラムはロジステロと言われています）。

画面の切り替え(場面設定)

画面の切り替えは、複数の画面を持つシステムを作る場合に重要になってきます。Windowsのシステムのように画面が重なり合うことが可能なタイプの画面出力システムを「オーバーラッピングウィンドウ」などと呼んだりもします。高解像度であれば問題なく、現実的なデスクトップを再現できますが、GBAのような低解像度では明示的な画面の切り替えシステムが必要になります。今回のプログラムでは十字キーの左右キーを使って画面の切り替えを行なうようにしました。

Windowsでのオーバーラッピングウィンドウ（画面の囲み部分がオーバーラッピングしている）

Column

トレードオフ

すべての要件を高い次元で兼ね備えたシステムができれば問題ないのですが、コストの制約や現テクノロジーの不備によって、実現にはむずかしいものがあります。ただ、多少機能を落としてもシステム的に成立していれば、実用にあたっては問題ありません。このようにシステム成立のために各要件をバランスさせせることをトレードオフと言いますが、システム設計では重要です。ただし、「妥協の産物」に陥らないように気をつける必要があります。

Column

BONSAI－WAREとは？

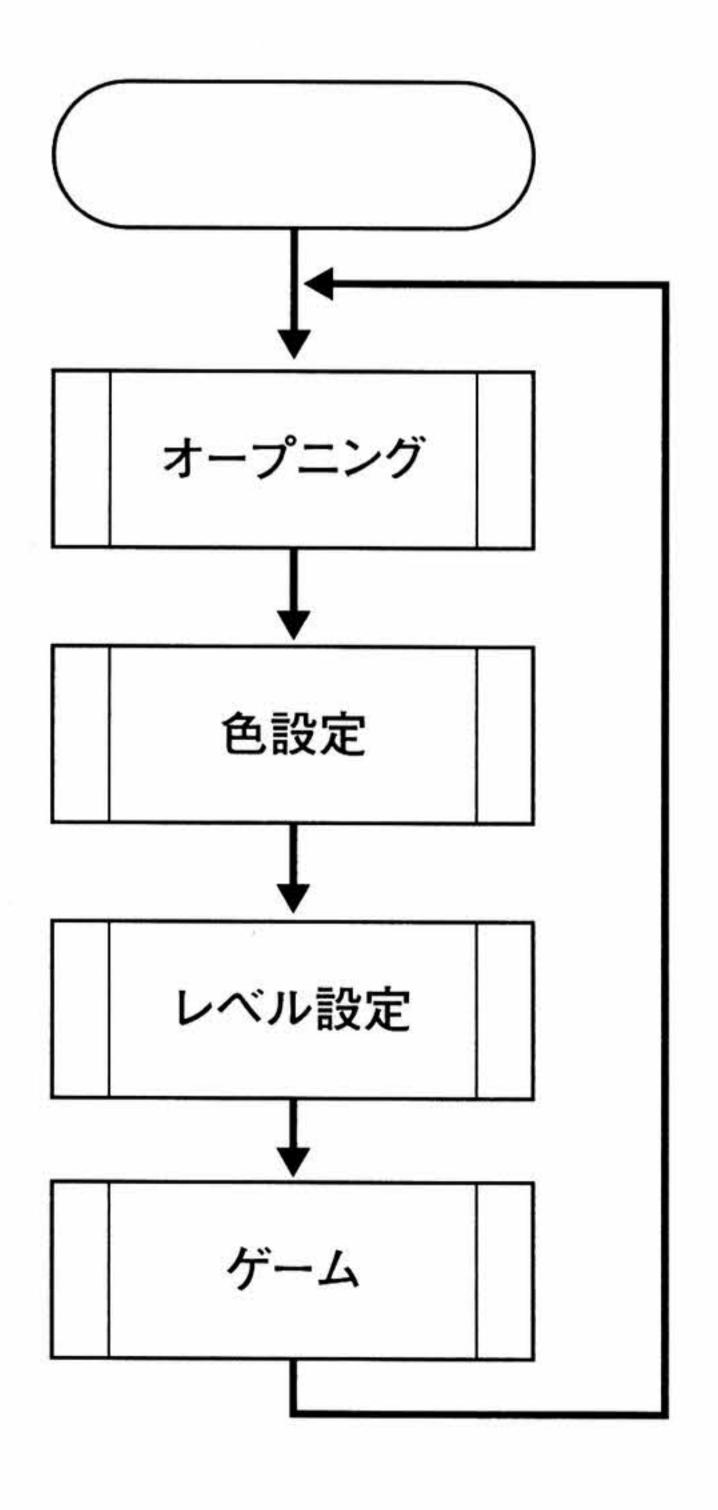
GBAは内蔵RAMにソフトをダウンロードして実行できます。この内蔵RAMは256KBあるので、プログラムの組み方次第でかなりのものが作成できます。実際に今回紹介するReversiAdevanceもプログラムサイズは56KB～68KBです(環境によって異なる)。ところで、この限られた容量で何かを行なう行為は何かに似ていないでしょうか？…そう、日本古来の伝統的な園芸「盆栽」です。ムダに伸びた枝を刈り込み整えていくやり方は、メモリを削り、限られた容量で行なうプログラミングとまったく同じです。まさに、ジジイが多いTeamKNOxの本領発揮というところでしょう!!

「盆栽」という言葉はすでに国際的に通用する言葉です。「BONSAI」で検索すると「BONSAI」そのものの説明は出てこないで、いきなり核心からはじまります(日本の盆栽ページの言葉が変わっただけのページみたいです)。ハサミを「開発ツール」に持ち替えて、盆栽みたいなプログラミングに取り組んでみましょう。そこには小さな宇宙があるはずです!!

GameBoyAdvance プログラミングII

いよいよプログラムの作成です。付属CD-ROMには完成したプログラムと掲載しているすべてのソースが収録されています。ですから、ここではソースプログラムを読み解きながら、プログラムの組み立て方を身につけていきましょう。

プログラムの構造（メイン）

Chapter03で各場面の構成要素の分析を行ないました。プログラミング（実装）の単位もこれにしたがうと直感的にわかりやすいと思います。つまり、次のような構造になります。

① オープニング
② 色設定
③ レベル設定
④ ゲーム画面

これにそれぞれ、キー入力や画面の表示などを割り付けるわけです。これらのプログラムをビューと呼ぶことにします。最後にそれらを全体的にまとめ上げるための大枠のプログラムを作ることで、全体のシステムを構築します。

それでは、実際にプログラムを見てみましょう。フローチャートで大まかな流れはわかったと思います。気をつけなければいけないのは、「オープニング」「色指定」「レベル設定」「ゲーム」の順に動作が行なわれていき、最後に再び「オープニング」に戻ることです。各構成要素は別ファイルとして外部で定義しています。これらを踏まえてメインプログラムの動作を追っていきましょう。

ReversiMain.c

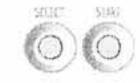
```
001: #include "GBA.h"
002:
003  #include "TeamKNOx/TeamKNOxLib.h"
004:
005: // Reversi constants
006: #include "ReversiConstants.h"
007:
008: extern u16 ViewOpening();
009: extern u16 ViewColorSelect();
010: extern u16 ViewLevelSelect();
011: extern u16 ViewGame();
012:
013  u16 gViewNumber;
014: u16 gMyColor;
015: u16 gGameLevel;
016:
017: // int main(void)
018: u16 AgbMain(void)
019: {
020:   gMyColor = 0;
021:   gGameLevel = 1;
022:
023:   SetMode( MODE_3 | BG2_ENABLE ); // Set MODE3
024:
025:   gViewNumber = KViewOpening;
```

```
026:     ViewOpening();
027:
028:     gViewNumber = KViewColorSelect;
029:     while(1){
030:       switch (gViewNumber){
031:         case KViewOpening:
032:           ViewOpening();
033:         case KViewColorSelect:
034:           ViewColorSelect();
035:           break;
036:         case KViewLevelSelect:
037:           ViewLevelSelect();
038:           break;
039:         case KViewGame:
040:           ViewGame();
041:           break;
042:
043:         default:
044:           ViewOpening();
045:           break;
046:       }
047:
048:     }
049:     return 0;
050:
051: }
052:
053: // EOF
```

●L.001：GBA.h のインクルード：

GBA の各種情報が記述されたヘッダーファイルの宣言です。

●L.003：TeamKNOxLib.h のインクルード：

オリジナルライブラリファイルのインクルード宣言です。

●L.006 ： ReversiConstants.h のインクルード(P69 参照)

リバーシプログラムのゲーム自体の定数が定義されたファイルです。

●L.008～L.011：外部(このファイル以外)で定義されたファイルの実行を宣言：

リバーシプログラムの各構成要素の使用を宣言します。

●L.013～L.015：グローバル変数の宣言：

gViewNumber はどの構成要素かを表わします。この変数を制御することによって実行させる構成要素（オープニングや色選択など）を変化させます。**gMyColor** は自分の石の色が黒か白か？ **gGameLevel** はゲームのレベルを表わします。

●L.018：メイン関数：

C 言語はよく、**main()** からはじまると言われています。一般的にはそうですが、各ターゲット（制御する対象物）と処理系によって異なります。たとえば Windows のプログラムでは WinMain がメイン関数になっています。GBA の場合も、今回の処理系は **AgbMain()** となっています。

●L.020～L.021：グローバル変数の初期化：

グローバル変数の初期化を行ないます。変数の初期化はプログラムの最初で行なうのが鉄則です。あるいはその変数を使うそばで行なうのもひとつの方法です。

●L.023：モードの初期化：

GBA のグラフィックモードの初期化を行ないます。今回はビットマップを利用するので、そのモードで設定します。

●L.025：ビューをオープニングにセット：

ビューの設定を最初の画面である「オープニング」に設定します。

●L.026：「オープニング」の実行：

オープニングを実行します。

●L.028：gViewNumber を色選択に変える：

「オープニング」を実行したのでビューを「色選択」に変えます。

●L.029：無限ループの作成：

このプログラムは無限ループしてます。while(1) を行なうことにより実現しています。

●L.030～L.041：動作の選択：

gViewNumber に応じた動作を行ないます。

●L.043：デフォルトの動作：

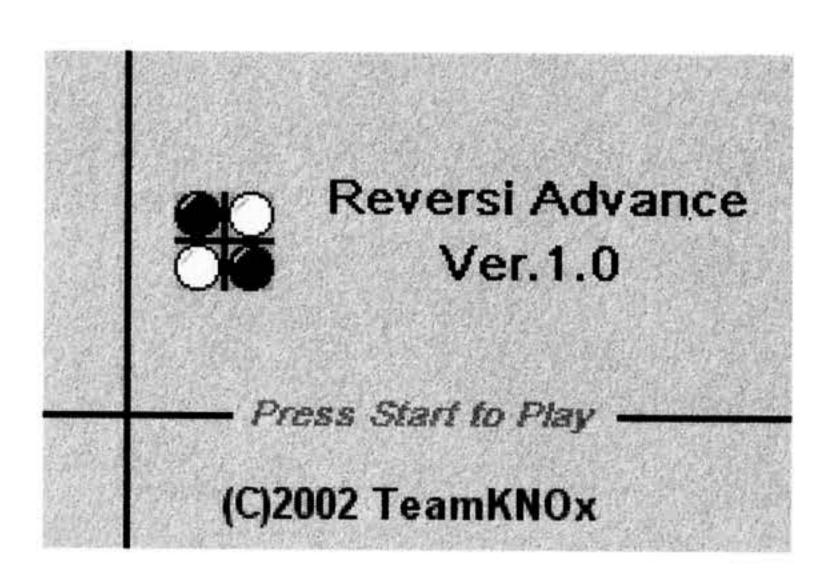
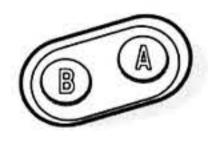
「オープニング」がデフォルトの動作になります。

●L.049：プログラムの復帰：

AgbMain()の戻り値になります。OS なしで動作している組み込みソフトのメインに戻り値は基本的には不要ですが、慣習にしたがってリターンしています。

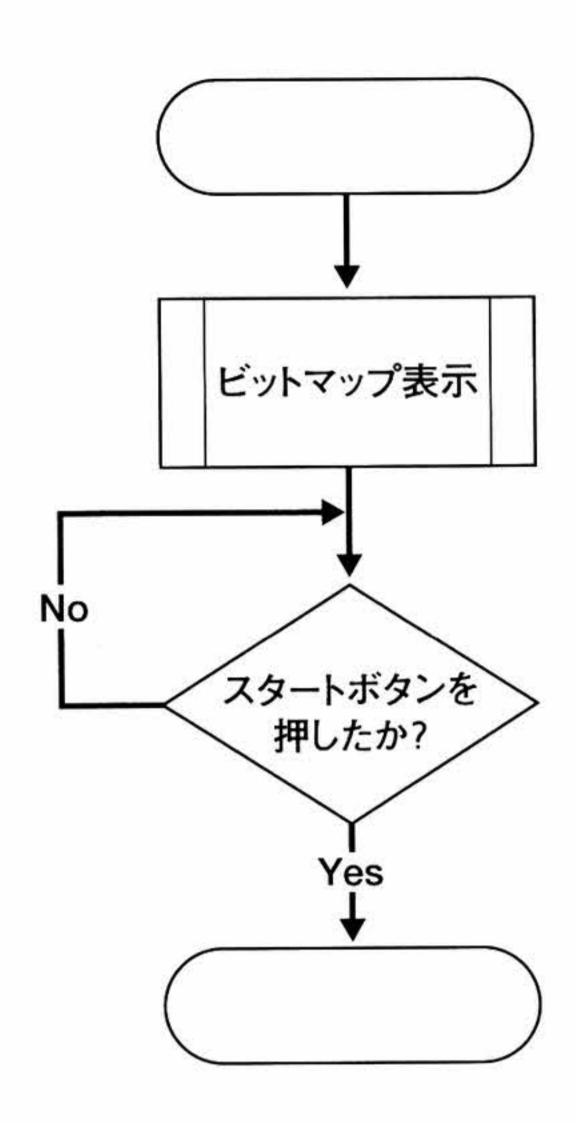
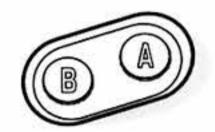
各構成要素の詳細分析

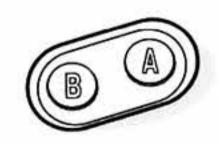
オープニング

オープニングはゲームのはじまりを告げるとともに、ユーザーに対してそのゲームを印象づけるための大切な部分です。さまざまなギミックが考えられますが、今回はシンプルにタイトル画面のみを表示させることにします。スタートボタンを押すことでゲーム開始とします。

4-2-1-1 オープニングの要求分析

くり返しになりますが、オープニングで求められるのは次の２つです。

① あらかじめ決められた画面（ビットマップファイル）を表示する

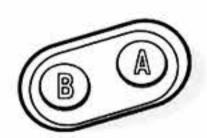
② スタートボタンが押されたらゲームを開始する

4-2-1-2 ビットマップファイルの表示

オープニングとは「プログラムの起動時」とほぼ同じ意味です。したがって、オープニングのタイトルはプログラムの起動時に画面に表示されればいいわけです。ビットマップファイルの表示はChapter03でも述べましたが、ビットマップファイルをあらかじめ配列として定義しておき、それをVRAMに書き込むことによって実現します。

4-2-1-3 スタートキーが押されたらゲーム開始

ボタンの読み取りは入力ポートに接続されたスイッチ（押しボタン）の読み取りです。スタートボタンはあるポートの特定のビットにアサインされています。これを読み取ることによってスタートボタンが押されたか否かを判断することができます。

では、ゲームパッドの読み取りについて具体的に見てみましょう。GBAで使われている各種入出力のアドレスはGBA.hに記述されています。GBA.hをテキストエディタで開くと「KEYS」という記述を見つけることができると思います。これが今回のプログラムで使われているゲームパッド部分のアドレスです。その上を少し見ると、「KEY_START」の記述を見つけることもできます。ここには、各キーの重み(位置)が記述されています。これらを前述のように組み合わせることによって、ゲームパッドの読み取りを行なっているわけです。

インクルードされるGBA.h(部分)

```
226:  #define KEY_A          1
227:  #define KEY_B          2
228:  #define KEY_SELECT     4
229:  #define KEY_START      8
230:  #define KEY_RIGHT      16
231:  #define KEY_LEFT       32
232:  #define KEY_UP         64
233:  #define KEY_DOWN       128
234:  #define KEY_R          256
235:  #define KEY_L          512
236:
237:  // キー入力用
238:  #define KEYS (volatile u32*)0x04000130
```

4-2-1-4 プログラムを書いてみる

ここまでの内容を、さっそくプログラムに書いてみましょう。実際のプログラムはReversiOpening.cに書かれています。各ラインごとに追っていきましょう。プログラムは動作別に適宜改行を入れてあります。

●L.001 ： GBA.hのインクルード

GBAの各種定数（アドレスや解像度などGBA固有の定数）が書かれているGBA.hというヘッダファイルをインクルードし（読み込み）ます。いちどは見ておいたほうがいいでしょう。

ReversiOpening.c

```
001:  #include "GBA.h"
002:
```

●L.003 ： extTeamKNOxLib.h のインクルード

TeamKNOxで開発した各種関数群（ライブラリ）の外部宣言が書かれたヘッダファイルです。ライブラリであるTeamKNOxLibについては、次のChapter 05で詳しく解説します。

```
003:  #include "TeamKNOx/extTeamKNOxLib.h"
004:
005:  // Reversi constants
```

●L.006 ： ReversiConstants.h のインクルード

今回の対象であるリバーシプログラムのゲーム自体の定数が定義されています。たとえばリバーシは8x8の升目でゲーム盤が構成されていますが、それらの数なども定義されています。

```
006:  #include "ReversiConstants.h"
007:
008:  // Application Specfic part
009:  // Graphic(Bitmap) data
```

ReversiConstants.hの内容

```
001: // ReversiConstants.h
002:
003: #ifndef REVERSI_CONSTANTS
004: #define REVERSI_CONSTANTS
005:
006: #define BOARD_START_POSITION_X 12
007: #define BOARD_START_POSITION_Y 12
008: #define BOARD_GRID_SIZE        17
009: #define BOARD_GRID_NUMBER      8
010: #define BOARD_GRID_LENGTH BOARD_GRID_SIZE*BOARD_
     GRID_NUMBER
011: #define BOARD_GRID_LINE_WIDTH  1
012:
013: #define BOARD_COLOR            0x0200
014: #define BOARD_GRID_LINE_COLOR  0x00
015:
016: #define STONE_SIZE_XY          16
017:
018: #define SUB_TITLE_PANE_SIZE_X  80
019: #define SUB_TITLE_PANE_SIZE_Y  60
020:
021: #define CURSOR_SIZE_XY         16
022: #define CURSOR_FIXED_OFFSET    8
023:
024: #define STONE_BLACK            1
025: #define STONE_WHITE            0
026:
027: #define BG_COLOR               0x6318
028:
029: enum ViewStatus
030: {
```

```
031:        KViewOpening = 0,
032:        KViewColorSelect,
033:        KViewLevelSelect,
034:        KViewGame,
035:        KViewGameEnd
036:
037: } gViewStatus;
038:
039: #define ISSIKI                    0
040:
041: #define CURSOR_FINGER             1
042: #define CURSOR_HOUR_GLASS         0
043:
044:
045: // use off screen
046: #define OFF_SCREEN 1
047:
048:
049: #if OFF_SCREEN
050:     #define OFF_SCREEN_ADDRESS    0x2020000
051: #else
052:     #define OFF_SCREEN_ADDRESS    VRAM_ADDRESS
053: #endif
054:
055:
056: #define TRANSPARENT_ON      0
057: #define TRANSPARENT_OFF     1
058:
059:
060: #define MULTIBOOT volatile const u8 __gba_multiboot;
061: MULTIBOOT
062:
063: #endif
```

●L.010 : opening.c のインクルード

ビットマップデータをCの配列ファイルに変換したものです。後述しますが、圧縮データを使っています。

```
010:  #include "bitmaps/opening.c"
011:
012:
013:
```

●L.014 : グローバル変数、使用の宣言

外部で定義されたグローバル変数 gViewNumber の使用の宣言。

```
014:  extern u16 gViewNumber;
```

●L.015 : 関数の開始

ViewOpening() の処理の開始

```
015:  u16 ViewOpening()
016:  {
```

●L.017～L.019 : ローカル変数の宣言

関数内で使うローカル変数を宣言します。keyWk は現在のキー(ボタン)の状態を、lastKeyWk は1つ前のキーの状態を保持します。stayThisView はループの制御に用います。

```
017:      u16 keyWk;          // current key data
018:      u16 lastKeyWk;      // prevous key data
019:      u16 stayThisView;
020:
```

●L.021 ：画面表示関数

(u16*)opening_Mapに格納されたデータを画面に表示します。実際の画面表示はこの関数によって行なわれています。

```
021:     BitBltMaskedComp(0, 0, SCREEN_SIZE_X, SCREEN_SIZE_
    Y, (u16*)opening_Map, OFF_SCREEN_ADDRESS);
022:
```

●L.023～L.025 ：プリプロセッサ・ディレクティブマクロ

これはプリプロセッサ・ディレクティブマクロと言います。マクロの定数であるOFF_SCREENが真であるときにOff2VRAM()がコンパイルされます。OFF_SCREENはReversiConstants.hに定義されています。

```
023: #if OFF_SCREEN
024:         Off2VRAM(OFF_SCREEN_ADDRESS, VRAM_ADDRESS);
025: #endif
026:
027:
```

●L.028～L.029 ：変数の初期化

lastKeyWkとkeyWkには、それぞれ現在と1つ前のボタンの情報が格納されています。stayThisViewはwhileの制御に使います。この変数が真であり続ける限り、プログラムはこのViewにとどまるわけです。

```
028:     lastKeyWk = keyWk = 0;
029:     stayThisView = 1;
```

●L.030～L.052：ループ

少し範囲が大きくなりましたが、この部分はブロックとして認識したほうがいいでしょう。**while(stayThisView)**のstayThisViewは初期化で1になっています。これは真なので無限ループになります。

```
030:     while(stayThisView){
031:
032:         keyWk = *KEYS;
033:
034:         if(lastKeyWk ^ keyWk){
035:             if(!(*KEYS & KEY_UP)){
036:             }
037:             if(!(*KEYS & KEY_DOWN)){
038:             }
039:             if(!(*KEYS & KEY_LEFT)){
040:             }
041:             if(!(*KEYS & KEY_RIGHT)){
042:             }
043:             if(!(*KEYS & KEY_A)){
044:                 stayThisView = 0;
045:             }
046:             if(!(*KEYS & KEY_START)){
047:                 stayThisView = 0;
048:             }
049:         }
050:
051:         lastKeyWk = keyWk;
052:         WaitForVsync(); // Wait VBL
053:
054:     }
055:
```

そのループの中で行なっていることは、次のようになります。

● L.032 - 1.ボタンの読み取り

● L.034 - 2.ボタンの変化の認識

● L.043 - 3.特定ボタンの変化の認識

ここで特定のボタンとして「スタート」あるいは「A」ボタンが押されると stayThisView = 0 を実行してループから脱出します。

● L.051 - 4.ボタン情報の更新

ボタンの評価が終わったら、ボタン情報の更新をします。具体的には lastKeyWk に現在のボタン情報である keyWk の内容を代入します。

● L.052 - 5.垂直帰線割り込みの終了待ち

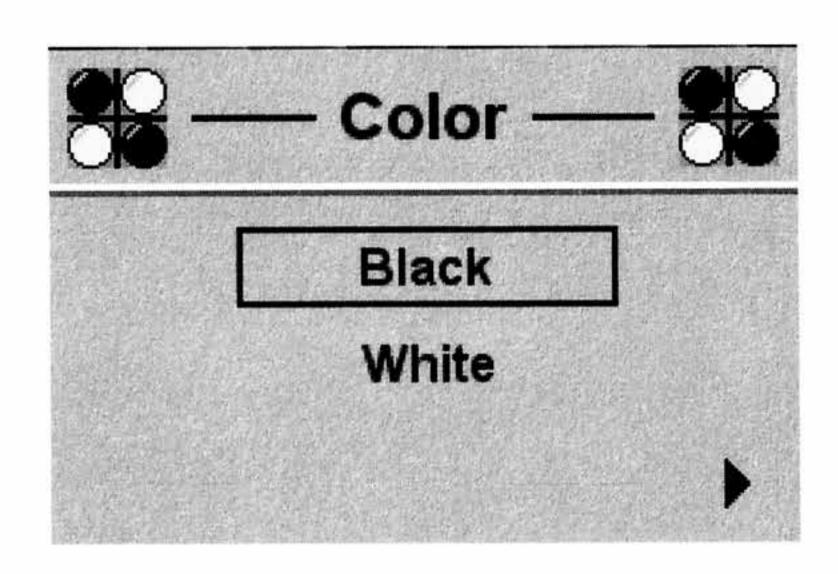
画面のリフレッシュが終了するまで待ちます。

● L.056 ～ L.058 ：関数の終了

gViewNumber を更新します(色選択画面に遷移する)。また、戻り値(現在は使っていない)があるので、リターンでこの関数は終了となります。

```
056:        gViewNumber = KViewColorSelect;
057:        return 0;
058:    }
059:
060:    // EOF
```

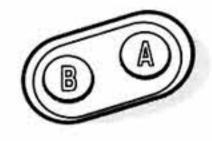
色設定

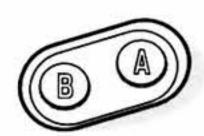
ここからがリバーシの実質的なプログラムと言ってもいいでしょう。色の選択は必須の機能です。

4-2-2-1 色設定の要求分析

例によって色設定の要求分析を行ないましょう。基本的なプログラム構成はオープニングに近いものがあり、次のようになります。

① あらかじめ決められた画面（ビットマップファイル）を表示

② 右ボタンが押されたらレベル設定画面

③ 上下ボタンでカーソルの移動

4-2-2-2 オープニングとの違い

基本的にはオープニングと違いはありませんが、いくつか機能が付加されています。それらをクローズアップすることによって効率的にプログラムの動作をトレースすることができます。大きな違いはカーソルを用いているところです。カーソルの描画と移動が新たに加わった点になります。

4-2-2-3 プログラムを書いてみる

では、実際のプログラムを見てみましょう。オープニングとの大きな違いだけを抜粋して解説していきます。プログラムはReversiColorSelect.cに記述されています。

ReversiColorSelect.c

```
001:  #include "GBA.h"
002:
003:  #include "TeamKNOx/extTeamKNOxLib.h"
004:
005:  // Reversi constants
006:  #include "ReversiConstants.h"
007:
008:  // Application Specfic part
009:  // Graphic(Bitmap) data
010:  #include "bitmaps/ColorSelection.c"
011:
012:
013:  extern u16 gViewNumber;
014:  extern u16 gMyColor;
015:
```

はじめの部分ではRevesiOpening.cと同様に、必要なヘッダファイルやライブラリをインクルードしています。ビットマップデータは、bitmaps/opening.cではなく、bitmaps/ColorSelection.cをインクルードしています。これも、やはりビットマップデータをCの配列ファイルに変換したもので、後述の圧縮データを使っています。

●L.016～L.021 ：カーソルの描画情報

カーソルを描画する上で必要な座標などのパラメータを定義します。定数などに具体的な名前をつけてプログラムの可読性を高めています。

```
016:  #define COLOR_SELECT_CURSOR_POS_X       60
017:  #define COLOR_SELECT_CURSOR_STEP        31
018:  #define COLOR_SELECT_CURSOR_OFFSET      58
019:  #define COLOR_SELECT_CURSOR_WIDTH       120
020:  #define COLOR_SELECT_CURSOR_HEIGHT      24
021:  #define COLOR_SELECT_CURSOR_COLOR       0x7C00
022:
```

●L.023～L.032 ：カーソル描画関数

drawCursorColorSelect()により、ライブラリ関数DrawBoxEmpty()が実行されます。この関数は、決められた座標に決められた色で四角形の箱を描画します。今回はカーソルである箱の大きさなどは一定なので、座標と色を変化するためのパラメータとして位置付けることにしました。

```
023:  void drawCusorColorSelect(u16 aCursorPosition, u16 aColor)
024:  {
025:
026:      DrawBoxEmpty(COLOR_SELECT_CURSOR_POS_X,
027:          aCursorPosition * COLOR_SELECT_CURSOR_STEP +
      COLOR_SELECT_CURSOR_OFFSET,
028:          COLOR_SELECT_CURSOR_WIDTH,
029:          COLOR_SELECT_CURSOR_HEIGHT,
030:          aColor,
031:          VRAM_ADDRESS);
032:  }
033:
```

●L.034～L.084：ViewColorSelect()

少し長いのですが、ひとまとまりになっているので、細切れではなく、一気に作成してしまいましょう。ボタンの入力を受け付け、それぞれの処理を行なうという、オープニング同様の構成になっているのがわかると思います。上下ボタンがカーソルの移動になります。上ボタンが押されればカーソルを上に移動し、下ボタンが押されれば下に移動します。Aあるいは右ボタンが押された場合、変化が発生します。**while(stayThisView)**の**stayThisView**を0にしてボタン読み取りのループから脱出します。

```
034:  u16 ViewColorSelect()
035:  {
036:      u16 keyWk;                 // current key data
037:      u16 lastKeyWk;             // prevous key data
038:      u16 stayThisView;
039:      u16 cursorPosY;
040:
041:      BitBltMaskedComp(0, 0, SCREEN_SIZE_X, SCREEN_SIZE_
      Y, (u16*)ColorSelection_Map, OFF_SCREEN_ADDRESS);
042:
043:  #if OFF_SCREEN
044:      Off2VRAM(OFF_SCREEN_ADDRESS, VRAM_ADDRESS);
045:  #endif
046:
047:      cursorPosY = gMyColor;
048:
049:      lastKeyWk = keyWk = 0;
050:      stayThisView = 1;
051:      while(stayThisView){
052:
053:          keyWk = *KEYS;
```

```
054:
055:        if(lastKeyWk ^ keyWk){
056:            drawCusorColorSelect(cursorPosY, BG_COLOR);
057:
058:            if(!(*KEYS & KEY_UP)){
059:                if(cursorPosY > 0){
060:                    cursorPosY--;
061:                }
062:            }
063:            if(!(*KEYS & KEY_DOWN)){
064:                if(cursorPosY < 1){
065:                    cursorPosY++;
066:                }
067:            }
068:            if(!(*KEYS & KEY_LEFT)){
069:            }
070:            if(!(*KEYS & KEY_RIGHT)){
071:                stayThisView = 0;
072:            }
073:            if(!(*KEYS & KEY_A)){
074:                stayThisView = 0;
075:            }
076:
077:            drawCusorColorSelect(cursorPosY, COLOR_SELECT
_CURSOR_COLOR);
078:        }
079:
080:        lastKeyWk = keyWk;
081:
082:        WaitForVsync(); // Wait VBL
083:
084:    }
```

●L.085 ：ビューの変化

ビューの管理をしている変数をレベル選択にします。

```
085:        gViewNumber = KViewLevelSelect;
```

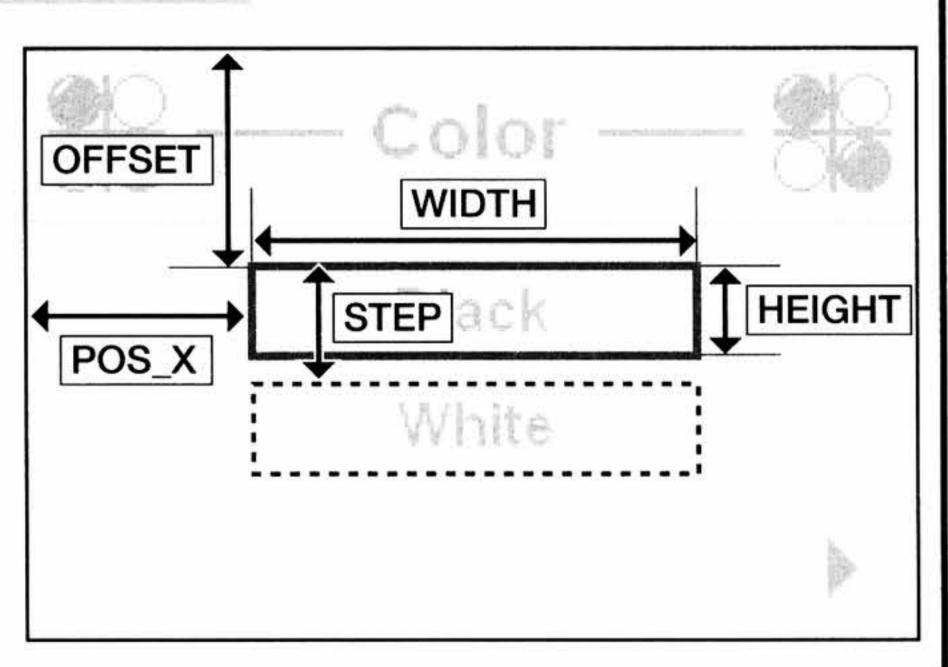
●L.086 ：自石の色設定

色を設定します。先程の色設定のために使っているカーソルのポジションを使います。

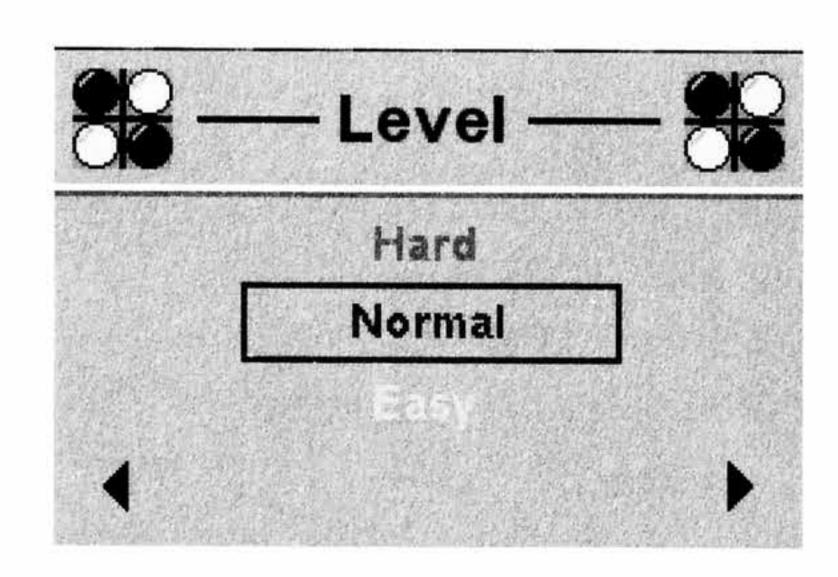
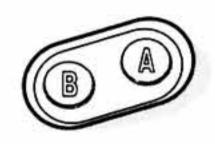
```
086:        gMyColor = cursorPosY;
087:
088:        return 0;
089:    }
090:
091:    // EOF
```

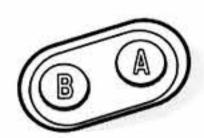
Column

カーソルの描画情報の読み方

変数名だけで見当がつくと思いますが、右図のように設定しています。描画サイズや座標は、ペイントソフト上で測定しておくといいでしょう。

レベル設定

コンピュータのリバーシの強さは、先読みレベルとその評価関数によって決まります。評価関数の設定は実装する人の好み次第ですが、これの味付けで強さが変化します。ただ、絶対的なものではなく、あくまでも相対的なものです。

いっぽう先読みレベルは単純に何手先を読むかにかかってきます。先読みの深さを深く取れば強くなっていきますが、それに伴って時間もかかります。しかし、ゲームをするプレイヤーのレベルもさまざまで、スタイルも色々です。時間がかかっても強いプログラムと勝負したい人もいれば、暇つぶしにちょっとやりたい場合などです。そういった人たちの要望をある程度吸収するためにも、レベル設定は不可欠の機能といえます。

4-2-3-1 レベル設定の要求分析

ほとんど、色設定のプログラムと変わりません。選択項目が２項目（黒・白）から３項目（レベル１・２・３）になったくらいです。ですから、色設定の動作が理解できればとくに問題ないと思います。

4-2-3-2 色設定との違い

色設定との大きな違いは前の状態（色設定）へ戻れることです。つまりもういちど色設定することもできますし、問題がなければそのままゲームをはじめることができるようになっている点です。

4-2-3-3 プログラムを書いてみる

すべて解説するのは冗長になるので、前述の大きな違いだけを見てみましょう（L.001～L.067までは、L.047の変数名がgMyColorからgGameLevelに変わるだけで同一です）。

色設定に戻るための操作は、十字ボタンの左とBボタンで行ないます。それらはそれぞれ、L.068～L.071、L.080～L.83に記述されています。とくにL.069とL.081にはビューを変化させるための具体的な代入のあることがわかると思います。

ReversiLevelSelect.c

```
068:        if(!(*KEYS & KEY_LEFT)){
069:            gViewNumber = KViewColorSelect;
070:            stayThisView = 0;
071:        }
072:        if(!(*KEYS & KEY_RIGHT)){
073:            gViewNumber = KViewGame;
074:            stayThisView = 0;
075:        }
076:        if(!(*KEYS & KEY_A)){
077:            gViewNumber = KViewGame;
078:            stayThisView = 0;
079:        }
```

```
080:                if(!(*KEYS & KEY_B)){
081:                    gViewNumber = KViewColorSelect;
082:                    stayThisView = 0;
083:                }
084:
085:                drawCusorLevelSelect(cursorPosY, LEVEL_SELECT_
    CURSOR_COLOR);
086:            }
087:
088:            lastKeyWk = keyWk;
089:            WaitForVsync(); // Wait VBL
090:
091:        }
092:        gGameLevel = cursorPosY;
093:
094:        return 0;
095:    }
096:
097:    // EOF
```

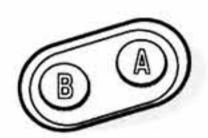
ゲーム

ゲーム部分は大きく分けて２つの部分から成り立っています。ひとつはエンジンと呼ばれる部分で、リバーシのルールや実際のプレイに伴う思考プログラムです。もうひとつはレンダラー（画面描画）です。それぞれのファイルは独立していて、ReversiEngine.cとReversiGame.cになります。これは仕様が変更になったときや、新しいエンジンに差し替えるのを容易にするためです。

実際、筆者の知人などは自分用の画面や独自の思考ルーチンを実装して楽しんでいますが、エンジン部分（ReversiEngine.c）はかなり独立性が高くなっています。これは元々Windows用に開発されたものを移植したものなので、他のプラットフォームへの移植も容易だと思います。

ただ、記述自体はGBAに限った内容ではないので、本書ではプログラムの説明は割愛します。このChapterの最後にプログラム全文を掲載しておきますので（またはCD-ROM収録のReversiEngine.cをテキストエディタで読み込んで）何をどうしているのか研究してみてください。思考ルーチンに興味がある人は、ぜひ参考文献を入手して参照してほしいと思います。

レンダラー

レンダラーは、エンジン部分が持っている盤面情報などを元に画面を描画するプログラムです。

4-2-5-1 レンダラーの要求分析

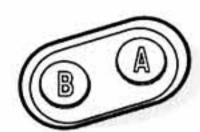
レンダラーの作成には、これまで学んだすべてのテクニック＋αが必要になります。レンダラーのプログラムに求められる主な要求内容は次のようになります。

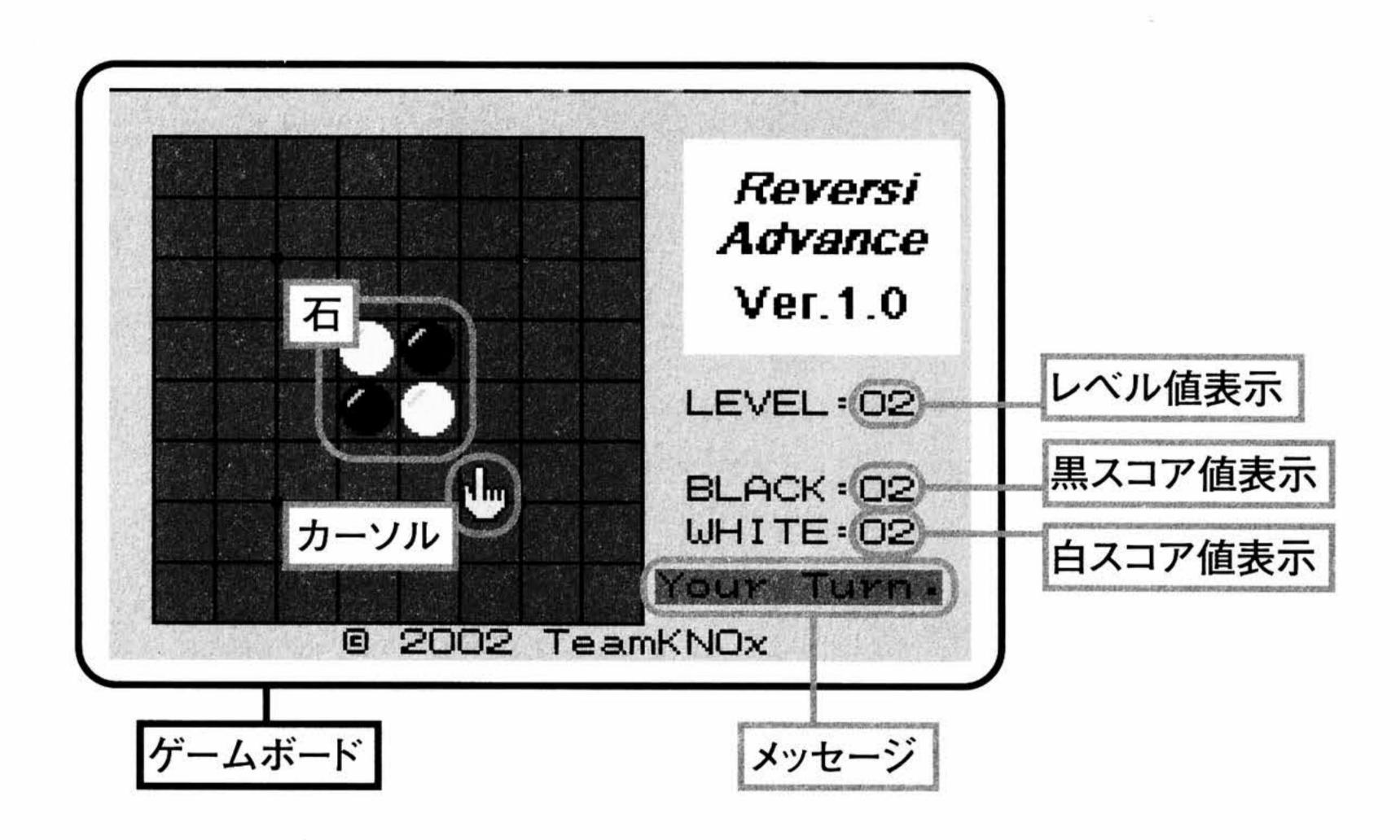
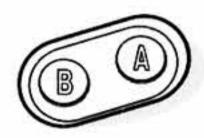
① あらかじめ決められた画面（ビットマップファイル）を表示

② 盤面情報の描画

③ カーソルの移動

④ カーソルの描画

⑤ 座標の指示

4-2-5-2 画面構成

ゲームの画面はいくつかの要素に分かれています。ここでは、それらの要素を見てみましょう。ゲームに必要な要素をピックアップしてみると、下の画面のようになります。

4-2-5-3 プログラムを書いてみる

作成するプログラム名はRversiGame.cです。L.001～L.006までは、これまでのプログラムと変わりません。

ReversiGame.c

```
001:  #include "GBA.h"
002:
003:  #include "TeamKNOx/extTeamKNOxLib.h"
004:
```

```
005:   // Reversi constants
006:   #include "ReversiConstants.h"
007:
008:   // Application Specfic part
009:   // Graphic(Bitmap) data
```

●L.010～L.019 ：グラフィックデータのインクルード

いままでのプログラムと大きく違うのが、このグラフィックデータのインクルードです。ここでは前述したグラフィックの要素を使います。それらのデータが配列に格納されています。

```
010:   #include "bitmaps/Stone_Black_16x16.c"
011:   #include "bitmaps/Stone_White_16x16.c"
012:
013:   #include "bitmaps/gameBoard.c"
014:
015:   #include "bitmaps/FingerCursorReady.c"
016:   #include "bitmaps/FingerCursorReady_mask.c"
017:
018:   #include "bitmaps/HourGlass.c"
019:   #include "bitmaps/HourGlass_mask.c"
020:
```

●L.021 ～L.031 ：外部変数の宣言

これらの変数はエンジン部分で使われているものです。これらの変数をレンダリングで使うために宣言しています。

```
021:   // Defined at ReversiEngine part...
022:   #include "ReversiEngine.h"
```

```
023:  extern u16 brdBaseInfo[BOARD_GRID_NUMBER][BOARD_
      GRID_NUMBER];
024:  extern void InitBaseInfo();
025:  extern u16 CheckPosition(u16* brd, u16 aStonePosX, u16
      aStonePosY, u16 aMyColor, u16 aWithReverse);
026:  extern u16 GameStatusCheck(u16 aNextColor);
027:  extern u16 ReversiEngine(u16 aComColor, u16 aComLevel);
028:
029:  extern u16 gViewNumber;
030:  extern u16 gMyColor;
031:  extern u16 gGameLevel;
032:
033:
```

●L.034～L.037 ： void SetUpBoard()

背景データを描画します。

```
034:  void SetUpBoard()
035:  {
036:      BitBltMaskedComp(0, 0, 240, 160, (u16*)gameBoard_Map,
      OFF_SCREEN_ADDRESS);
037:  }
038:
```

●L.039～L.071 ：
u16 SetUpAStone(u16 aStonePosX, u16 aStonePosY)

黒と白の石を描画します。ISSIKIモードがアクティブの場合は画面の石はすべて白になりますが、プログラム自身は黒石を内部で認識していま

す。これは囲碁の一色碁からヒントを得て実装したものです。囲碁と違い、リバーシは色の移り変わりが速いので非常にむずかしくなります。

```
039: u16 SetUpAStone(u16 aStonePosX, u16 aStonePosY)
040: {
041:     u16 positionX, positionY;
042:
043:     if(aStonePosY == 8){
044:         return 0;
045:     }
046:     else{
047:
048:     positionX = aStonePosX * BOARD_GRID_SIZE +
     BOARD_START_POSITION_X;
049:     positionY = aStonePosY * BOARD_GRID_SIZE +
     BOARD_START_POSITION_Y;
050:
051:     switch(brdBaseInfo[aStonePosX][aStonePosY]){
052:         case ID_STONE_NONE:
053:             break;
054:
055:             case ID_STONE_BLACK:
056:             if(ISSIKI)
057:                     BitBltMaskedComp(positionX, positionY,
     STONE_SIZE_XY, STONE_SIZE_XY,
     (u16*)Stone_White_16x16_Map, OFF_SCREEN_ADDRESS);
058:             else
059:                 BitBltMaskedComp(positionX, positionY,
     STONE_SIZE_XY, STONE_SIZE_XY,
     (u16*)Stone_Black_16x16_Map, OFF_SCREEN_ADDRESS);
060:             break;
061:
```

```
062:            case ID_STONE_WHITE:
063:                BitBltMaskedComp(positionX, positionY,
STONE_SIZE_XY, STONE_SIZE_XY,
(u16*)Stone_White_16x16_Map, OFF_SCREEN_ADDRESS);
064:                break;
065:
066:            default:
067:                break;
068:        }
069:        return 1;
070:        }
071: }
072:
073:
```

● L.074 ～ L.083 ： void SetUpStone()

盤面情報を元に黒･白の石を描画します。実際の描画は setUpAStone() が行ないます。

```
074: void SetUpStone()
075: {
076:     u16 i, j;
077:
078:     for(j = 0;j < BOARD_GRID_NUMBER;j++){
079:         for(i = 0;i < BOARD_GRID_NUMBER;i++){
080:             SetUpAStone(i, j);
081:         }
082:     }
083: }
084:
```

●L.085～L.113：

void CountUpScore(u8 *arBlackScore, u8 *arWhiteScore)

黒と白石の数をカウントします。

```
085: void CountUpScore(u8 *arBlackScore, u8 *arWhiteScore)
086: {
087:     u16 i, j;
088:     u8 blackScore, whiteScore;
089:
090:     blackScore = whiteScore = 0;
091:
092:     for(j = 0;j < BOARD_GRID_NUMBER;j++){
093:         for(i = 0;i < BOARD_GRID_NUMBER;i++){
094:             switch(brdBaseInfo[i][j]){
095:                 case ID_STONE_NONE:
096:                     break;
097:
098:                 case ID_STONE_BLACK:
099:                     blackScore++;
100:                     break;
101:
102:                 case ID_STONE_WHITE:
103:                     whiteScore++;
104:                     break;
105:
106:                 default:
107:                     break;
108:             }
109:         }
110:     }
111:     *arBlackScore = blackScore;
112:     *arWhiteScore = whiteScore;
```

```
113:  }
114:
```

●L.115～L.134 ： void SetUpInfo(u8 comLevel)

各種情報の表示を行ないます。表示項目はレベル、黒数、白数です。

```
115:  void SetUpInfo(u8 comLevel)
116:  {
117:      u8 blackScore, whiteScore;
118:
119:      blackScore = whiteScore = 0;
120:
121:      CountUpScore(&blackScore, &whiteScore);
122:
123:      DrawText(BOARD_START_POSITION_X * 2 + BOARD_GRID
      _LENGTH, BOARD_START_POSITION_Y* 2 + SUB_TITLE_PANE
      _SIZE_Y, "LEVEL:", RGB(0, 0, 0), BG_COLOR,TRANSPARENT_
      OFF, OFF_SCREEN_ADDRESS);
124:      num02str(comLevel + 1);
125:      DrawText(BOARD_START_POSITION_X * 2 + BOARD_GRID_
      LENGTH + 8 * 6, BOARD_START_POSITION_Y * 2 + SUB_TITLE
      _PANE_SIZE_Y, gWorkStr, RGB(0, 0, 0), BG_COLOR,
      TRANSPARENT_OFF, OFF_SCREEN_ADDRESS);
126:
127:      DrawText(BOARD_START_POSITION_X * 2 + BOARD_GRID_
      LENGTH, BOARD_START_POSITION_Y* 4 + SUB_TITLE_PANE_
      SIZE_Y, "BLACK:", RGB(0, 0, 0), BG_COLOR, TRANSPARENT_
      OFF, OFF_SCREEN_ADDRESS);
128:      num02str(blackScore);
129:      DrawText(BOARD_START_POSITION_X * 2 + BOARD_GRID_
```

```
     LENGTH + 8 * 6, BOARD_START_POSITION_Y* 4 + SUB_TITLE_
     PANE_SIZE_Y, gWorkStr, RGB(0, 0, 0), BG_COLOR,
     TRANSPARENT_OFF, OFF_SCREEN_ADDRESS);
130:
131:     DrawText(BOARD_START_POSITION_X * 2 +
     BOARD_GRID_LENGTH, BOARD_START_POSITION_Y * 5 +
     SUB_TITLE_PANE_SIZE_Y, "WHITE:", RGB(0, 0, 10),
     BG_COLOR, TRANSPARENT_OFF, OFF_SCREEN_ADDRESS);
132:     num02str(whiteScore);
133:     DrawText(BOARD_START_POSITION_X * 2 + BOARD_GRID_
     LENGTH + 8 * 6, BOARD_START_POSITION_Y* 5 + SUB_TITLE_
     PANE_SIZE_Y, gWorkStr, RGB(0, 0, 0), BG_COLOR,
     TRANSPARENT _OFF, OFF_SCREEN_ADDRESS);
134: }
135:
```

●L.136～L.140 ： void SetUpMessage(u8* aStrings)

メッセージ部分に文字列を表示させます。

```
136: void SetUpMessage(u8* aStrings)
137: {
138:     DrawBox(BOARD_START_POSITION_X * 2 + BOARD_GRID_
     LENGTH - 8, BOARD_START_POSITION_Y* 6 + SUB_TITLE_
     PANE_SIZE_Y + 2, 80, 10, 0x001f, OFF_SCREEN_ADDRESS);
139:     DrawText(BOARD_START_POSITION_X * 2 + BOARD_GRID_
     LENGTH - 8, BOARD_START_POSITION_Y * 6 + SUB_TITLE_
     PANE_SIZE_Y + 3, aStrings, RGB(0, 0, 0), 0x001F, TRANSPARENT
     _OFF, OFF_SCREEN_ADDRESS);
140: }
141:
142: u16 gBGSaveData[16][16];
```

●L.144～L.159：

void SetUpCursor(u16 aPositionX, u16 aPositionY, u16 aGameTurn)

カーソル表示を行ないます。カーソルポジションは0～7の8段階で、それを実際の画面の表示座標に変換します。また、手番(自番/相手番)によって、指カーソル(自番)にしたり砂時計カーソル(相手番：思考中)にします。

```
143:
144: void SetUpCursor(u16 aPositionX, u16 aPositionY, u16 aGameTurn)
145: {
146:     u16 positionX, positionY;
147:
148:     positionX = aPositionX * BOARD_GRID_SIZE + BOARD_
     START_POSITION_X;
149:     positionY = aPositionY * BOARD_GRID_SIZE + BOARD_
     START_POSITION_Y + CURSOR_FIXED_OFFSET;
150:
151:     if(aGameTurn){
152:         BitBltMasked(positionX, positionY, CURSOR_SIZE_XY,
     CURSOR_SIZE_XY, (u16*)FingerCursorReady_Map, (u16*)
     FingerCursorReady_mask_Map, OFF_SCREEN_ADDRESS);
153:         //  BitBltMaskedComp(positionX, positionY, CURSOR_
     SIZE_XY, CURSOR_SIZE_XY, (u16*)FingerCursorReady_Map,
     OFF_SCREEN_ADDRESS);
154:     }
155:     else{
156:         BitBltMasked(positionX, positionY, CURSOR_SIZE_XY,
     CURSOR_SIZE_XY, (u16*)HourGlass_Map, (u16*)HourGlass_
     mask_Map, OFF_SCREEN_ADDRESS);
157:     }
158:
159: }
```

●L.161〜L.178：

void SaveSprite(u16 aPositionX, u16 aPositionY, u32 aVRAM)

VRAM の特定の領域を退避します。

```
160:
161: void SaveSprite(u16 aPositionX, u16 aPositionY, u32 aVRAM)
162: {
163:     u16 i, j;
164:     u16 positionX, positionY;
165:     u16* ScreenBuffer;
166:
167:     positionX = aPositionX * BOARD_GRID_SIZE + BOARD_
     START_POSITION_X;
168:     positionY = aPositionY * BOARD_GRID_SIZE + BOARD_
     START_POSITION_Y + CURSOR_FIXED_OFFSET;
169:
170:     ScreenBuffer = (u16*)aVRAM;
171:
172:     for(j = 0;j < 16;j++){
173:         for(i = 0;i < 16;i++){
174:             gBGSaveData[i][j] = ScreenBuffer[(positionY + j) *
     SCREEN_SIZE_X + positionX + i];
175:         }
176:     }
177:
178: }
179:
```

●L.180～L.197：

void RestoreSprite(u16 aPositionX, u16 aPositionY, u32 aVRAM)

SaveSpriteで退避した領域をVRAMへ戻します。

```
180: void RestoreSprite(u16 aPositionX, u16 aPositionY, u32 aVRAM)
181: {
182:     u16 i, j;
183:     u16 positionX, positionY;
184:     u16* ScreenBuffer;
185: 
186:     ScreenBuffer = (u16*)aVRAM;
187: 
188:     positionX = aPositionX * BOARD_GRID_SIZE + BOARD_
     START_POSITION_X;
189:     positionY = aPositionY * BOARD_GRID_SIZE + BOARD_
     START_POSITION_Y + CURSOR_FIXED_OFFSET;
190: 
191:     for(j = 0;j < 16;j++){
192:         for(i = 0;i < 16;i++){
193:             ScreenBuffer[(positionY + j) * SCREEN_SIZE_X +
     positionX + i] = gBGSaveData[i][j];
194:         }
195:     }
196: 
197: }
198: 
199: 
```

●L.200～L.397 ：u16 ViewGame()

この部分がレンダラーの中心となるところです。そのため、サイズもこれまでのプログラムに比べて大きくなっています。基本的には、これまでのプログラムと同じで、カーソルを動かして、A ボタンで決定します。ただ、処理すべき内容が増えたのでプログラムのライン数も増えたわけです。

それでは、プログラムをもう少し詳しくみていきましょう。

●L.202～L.211 ：変数の宣言

関数 ViewGame()で使用する変数を宣言している部分です。

```
200:  u16 ViewGame()
201:  {
202:      u16 keyWk;                // current key data
203:      u16 lastKeyWk;            // prevous key data
204:      u16 stayThisView;
205:
206:      u16 cursorPosX, cursorPosY;
207:
208:      u16 myColor, comColor, gameResult;
209:      u8 blackScore, whiteScore, comLevel;
210:      u16 gameTurn;
211:      u8 put;
212:
```

●L.213～L.265 ：初期化を行なう

変数と画面構成の初期化を行ないます。変数の初期化は、スコアやゲームのモードなどを保持している変数に対して行ないます。また、画面構成の初期化はカーソルの位置や最初に置いておく石などの設定です。

```
213:        gameResult = GAME_STATUS_NORMAL;
214:
215:        myColor = ID_STONE_BLACK;
216:        comColor = ID_STONE_WHITE;
217:        gameTurn = GAME_TURN_WHITE;
218:        switch(gMyColor){
219:            case 0:
220:                myColor = ID_STONE_BLACK;
221:                comColor = ID_STONE_WHITE;
222:                gameTurn = GAME_TURN_WHITE;
223:                break;
224:            case 1:
225:                myColor = ID_STONE_WHITE;
226:                comColor = ID_STONE_BLACK;
227:                gameTurn = GAME_TURN_BLACK;
228:                break;
229:            default:
230:                break;
231:        }
232:
233:        comLevel = 1;
234:        switch(gGameLevel){
235:            case 0:
236:                comLevel = 2;
237:                break;
238:            case 1:
239:                comLevel = 1;
240:                break;
241:            case 2:
242:                comLevel = 0;
243:                break;
244:            default:
```

```
245:            break;
246:    }
247:
248:
249:    blackScore = whiteScore = 0;
250:
251:    cursorPosX = cursorPosY = BOARD_GRID_NUMBER / 2 + 1;
252:
253:    InitBaseInfo();
254:
255:    SetUpBoard();
256:    SetUpStone();
257:    SetUpInfo(comLevel);
258:    SaveSprite(cursorPosX, cursorPosY, OFF_SCREEN_ADDRESS);
259:    SetUpMessage(" Start....");
260:
261:    lastKeyWk = keyWk = 0;
262:    stayThisView = 1;
263:
264:    gameTurn = 0;
265:    put = 0;
```

● L.268 ～ L.309 ：自番のカーソル動作のサポート

自分の手番でのカーソルの動きをサポートします。十字ボタンやＡボタンの入力を受け付けてゲームは進みますが、カーソルが動く直前にカーソル下のVRAM情報をセーブし、動いたあとに先ほどセーブした内容を再描画します。

盤面上の石は独立して保持され、レンダリングされるので、カーソル位置とカーソルがずれている部分に合わせてSetUpAStone()を実行します。

Ａボタンが押されたときは、そのカーソル位置に自石がおけるかどうか

をCheckPosition()でチェックして、置ける場合はput = 0で相手番に変わります。

```
266:     while(stayThisView)
267:     {
268:         keyWk = *KEYS;
269:         if (gameTurn == myColor - 1)
270:         {
271:             SetUpMessage("Your Turn.");
272:             if(!(lastKeyWk ^ keyWk)){
273:                 continue;
274:             }
275:             lastKeyWk = keyWk;
276:             RestoreSprite(cursorPosX, cursorPosY, OFF_SCREEN
_ADDRESS);
277:             SetUpAStone(cursorPosX, cursorPosY);
278:             SetUpAStone(cursorPosX, cursorPosY + 1);
279:             if(!(*KEYS & KEY_UP)){
280:                 if(cursorPosY > 0){
281:                     cursorPosY--;
282:                 }
283:             }
284:             if(!(*KEYS & KEY_DOWN)){
285:                 if(cursorPosY < BOARD_GRID_NUMBER - 1){
286:                     cursorPosY++;
287:                 }
288:             }
289:             if(!(*KEYS & KEY_LEFT)){
290:                 if(cursorPosX > 0){
291:                     cursorPosX--;
292:                 }
293:             }
```

```
294:            if(!(*KEYS & KEY_RIGHT)){
295:                if(cursorPosX < BOARD_GRID_NUMBER - 1){
296:                    cursorPosX++;
297:                }
298:            }
299:            if(!(*KEYS & KEY_A)){
300:                if(CheckPosition(&(brdBaseInfo[0][0]), cursorPosX,
    cursorPosY, myColor, 1)) {
301:                    put = 1;
302:                }
303:                SetUpStone();
304:                SetUpInfo(comLevel);
305:            }
306:            SetUpAStone(cursorPosX, cursorPosY);
307:            SaveSprite(cursorPosX, cursorPosY, OFF_SCREEN_
    ADDRESS);
308:            SetUpCursor(cursorPosX, cursorPosY, CURSOR_
    FINGER);
309:        }
```

●L.310～L.324 ：相手番のカーソル動作のサポート

プログラムの手番に切り替わったときの処理です。カーソルを砂時計に変更してメッセージ領域に"Thinking.."を表示し、エンジン部分を呼び出します。エンジン部分の処理後にput＝1で着手したことを示します。

```
310:        else if (gameTurn == comColor - 1)
311:        {
312:            SetUpCursor(cursorPosX, cursorPosY, CURSOR_
    HOUR_GLASS);
```

```
313:                SetUpMessage("Thinking..");
314:                RestoreSprite(cursorPosX, cursorPosY, OFF_SCREEN
    _ADDRESS);
315:
316:                #if OFF_SCREEN
317:                    Off2VRAM(OFF_SCREEN_ADDRESS, VRAM_
    ADDRESS);
318:                #endif
319:
320:                ReversiEngine(comColor, comLevel*2);
321:                SetUpStone();
322:                SetUpInfo(comLevel);
323:                put = 1;
324:            }
```

● L.325 ～ L.387 ：現在のゲームのステータスに応じた処理

現在のゲームのステータスに応じた処理を行なう部分です。ゲームのステータスには次の種類があり、以下の処理を行ないます。

① GAME_STATUS_NORMAL

自番と相手番の手番を変えます。

② GAME_STATUS_END

カーソルを砂時計に変更します。スコアを測定して、どちらが勝ったか("Black Win!!" または "White Win !!")あるいは引き分けか("Draw !!")の表示を行ないます。また、メッセージ領域に "Game End" の表示を行ないます。ビューをオープニングに戻してループから抜けます。

③ GAME_STATUS_BLACK_PASS

黒手番がパスのときに実行します。"Black Pass" の表示を行ない、ボタンの入力待ちになります。

④ GAME_STATUS_WHITE_PASS

白手番がパスのときに実行します。"White Pass" の表示を行ない、ボタンの入力待ちになります。

```
325:            if (put == 1) {
326:                put = 0;
327:                gameResult = GameStatusCheck((!gameTurn) + 1);
328:                switch(gameResult){
329:                case GAME_STATUS_NORMAL:
330:                    gameTurn = !gameTurn;
331:                    break;
332:                case GAME_STATUS_END:
333:                    // Game End
334:                    SetUpCursor(cursorPosX, cursorPosY, CURSOR_
        HOUR_GLASS);
335:                    SetUpMessage(" Game End");
                        DrawBoxHalf(BOARD_START_POSITION_X / 2,
        BOARD_START_POSITION_Y + BOARD_GRID_SIZE * 3,
        BOARD_START_POSITION_X + BOARD_GRID_SIZE * 8,
336:    BOARD_GRID_SIZE * 2, OFF_SCREEN_ADDRESS);
337:
338:                    CountUpScore(&blackScore, &whiteScore);
339:                    if(blackScore > whiteScore)
340:                        DrawText(BOARD_START_POSITION_X +
        BOARD_GRID_SIZE, BOARD_START_POSITION_Y + BOARD_
        GRID_SIZE * 4, "Black Win !!", RGB(30, 30, 30), 0x7FFF,
        TRANSPARENT_ON, OFF_SCREEN_ADDRESS);
```

```
341:                if(blackScore < whiteScore)
342:                    DrawText(BOARD_START_POSITION_X +
BOARD_GRID_SIZE, BOARD_START_POSITION_Y + BOARD_
GRID_SIZE * 4, "White Win !!", RGB(30, 30, 30), 0x7FFF,
TRANSPARENT_ON, OFF_SCREEN_ADDRESS);
343:                if(blackScore == whiteScore)
344:                    DrawText(BOARD_START_POSITION_X +
BOARD_GRID_SIZE * 2, BOARD_START_POSITION_Y +
BOARD_GRID_SIZE * 4, "Draw !!", RGB(30, 30, 30), 0x7FFF,
TRANSPARENT_ON, OFF_SCREEN_ADDRESS);
345:            #if OFF_SCREEN
346:                Off2VRAM(OFF_SCREEN_ADDRESS, VRAM_
ADDRESS);
347:            #endif
348:
349:                gViewNumber = KViewOpening;
350:                stayThisView = 0;
351: //         #if OFF_SCREEN
352: //             Off2VRAM(OFF_SCREEN_ADDRESS, VRAM_
ADDRESS);
353: //         #endif
354:                AWait();
355:                break;
356:            case GAME_STATUS_BLACK_PASS:
357:                SetUpCursor(cursorPosX, cursorPosY, CURSOR_
HOUR_GLASS);
358:                SetUpMessage("Black Pass");
359:                DrawBoxHalf(BOARD_START_POSITION_X / 2,
BOARD_START_POSITION_Y + BOARD_GRID_SIZE * 3,
BOARD_START_POSITION_X + BOARD_GRID_SIZE * 8,
BOARD_GRID_SIZE * 2, OFF_SCREEN_ADDRESS);
360:                DrawText(BOARD_START_POSITION_X +
```

```
          BOARD_GRID_SIZE * 2, BOARD_START_POSITION_Y +
          BOARD_GRID_SIZE * 4, "Black Pass", RGB(30, 30, 30),
          0x7FFF, TRANSPARENT_ON, OFF_SCREEN_ADDRESS);
361:              #if OFF_SCREEN
362:                  Off2VRAM(OFF_SCREEN_ADDRESS, VRAM_
          ADDRESS);
363:              #endif
364:                  AWait();
365:                  SetUpBoard();
366:                  SetUpStone();
367:                  SetUpInfo(comLevel);
368:                  SetUpCursor(cursorPosX, cursorPosY, CURSOR_
          FINGER);
369:                  break;
370:              case GAME_STATUS_WHITE_PASS:
371:                  SetUpCursor(cursorPosX, cursorPosY, CURSOR_
          HOUR_GLASS);
372:                  SetUpMessage("White Pass");
373:                  DrawBoxHalf(BOARD_START_POSITION_X / 2,
          BOARD_START_POSITION_Y + BOARD_GRID_SIZE * 3,
          BOARD_START_POSITION_X + BOARD_GRID_SIZE * 8,
          BOARD_GRID_SIZE * 2, OFF_SCREEN_ADDRESS);
374:                  DrawText(BOARD_START_POSITION_X +
          BOARD_GRID_SIZE * 2, BOARD_START_POSITION_Y +
          BOARD_GRID_SIZE * 4, "White Pass", RGB(30, 30, 30),
          0x7FFF, TRANSPARENT_ON, OFF_SCREEN_ADDRESS);
375:              #if OFF_SCREEN
376:                  Off2VRAM(OFF_SCREEN_ADDRESS, VRAM_
          ADDRESS);
377:              #endif
378:                  AWait();
379:                  SetUpBoard();
```

```
380:                    SetUpStone();
381:                    SetUpInfo(comLevel);
382:                    SetUpCursor(cursorPosX, cursorPosY, CURSOR_
FINGER);
383:                    break;
384:                default:
385:                    break;
386:                }
387:            }
388:
```

●L.389～L.396 ： VRAMへ転送

VRAMへ転送します。垂直同期を待って画面をリフレッシュしたあと、最初に戻ります。

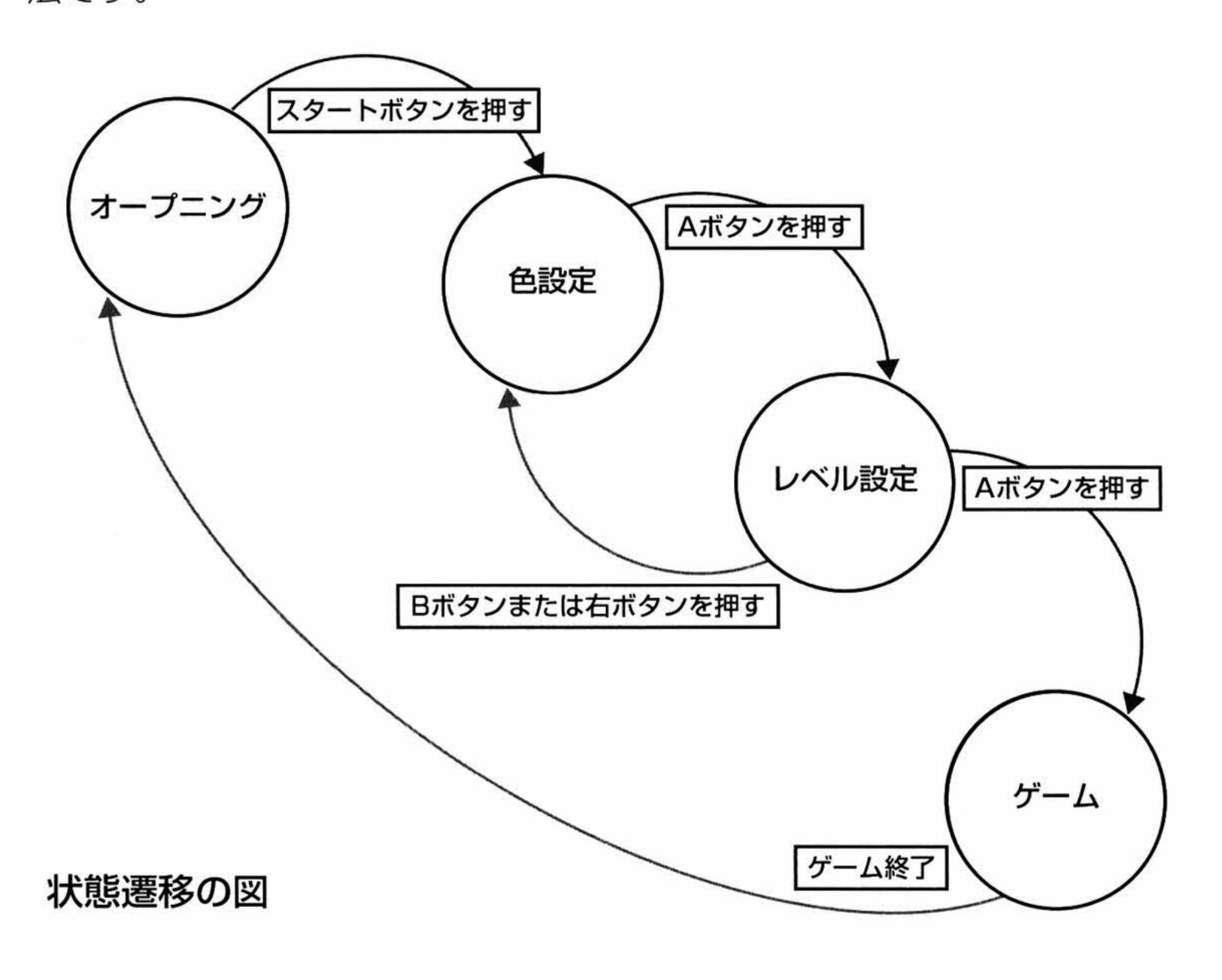
```
389:            #if OFF_SCREEN
390:                Off2VRAM(OFF_SCREEN_ADDRESS, VRAM_ADDRESS);
391:            #endif
392:
393:            WaitForVsync(); // Wait VBL
394:        }
395:
396:        return 0;
397:    }
398:
399:    // EOF
```

状態遷移

いままで見てきたプログラムは、「オープニング」➡「色選択」⬅➡「レベル選択」➡（「ゲーム」）とさまざまな状態を持つことがわかります。

また、これらの状態は変化することもわかります。これを「状態遷移」と言います。状態遷移はステートマシン（State Machine）とも呼ばれます。プログラムの構成や動きの移り変わりを把握するためには、大変便利な手法です。

オープニング
スタートボタンを押す
色設定
Aボタンを押す
レベル設定
Aボタンを押す
Bボタンまたは右ボタンを押す
ゲーム
ゲーム終了

状態遷移の図

問題点の確認

ここで制作したソフトの画面は、「オープニング」「色設定」「レベル設定」「ゲーム」の４画面になり、これらはフルビットマップで表示されます。この画面構成の容量を計算してみると次のようになります。

240 [pixel] x 160 [pixel] x 2 [Byte] = 76,800 [Byte]

ところで画面の枚数は４枚なので…

76,800 [Byte] x 4 [枚] = 307200 [Byte]

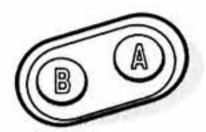
のようになり、これではデータを格納するだけでEWRAMの容量のすべてを消費してしまうことになります。そのため、これらの画像データは圧縮してプログラムを組む必要があることがわかります。

容量とスピードのトレードオフ

圧縮すればデータの容量は減るわけですが、良いことばかりではありません。圧縮されているデータを展開することを伸張と言います（解凍という言葉も慣用的に使われていますが、これは正しくありません）が、当然ながらこれには伸張する時間が必要になってきます。これがどの程度かかるかは非常に重要です。伸張するための時間がプログラムの実行時間に加わるためスピード感が失われる恐れが出てくるわけです。ただ、今回はメモリ容量の制約が圧倒的に大きいので、容量の削減を最優先で行なうものとします。

データ圧縮テクニックとプログラムの実装

これらの画像データは、あらかじめ圧縮した配列の形で保持するものとしました。具体的には専用の圧縮ツール(BmpCnv)を作り、それに本来のビットマップファイルの画像データを圧縮させました。

このツールはTeamKNOxのオリジナルツールで、筆者の知人が開発しコントリビュートしてくれました。BmpCnvが出力するデータに合わせたライブラリも開発しました。リスト中のBitBltMaskedComp()がこのデータに対応する関数になります。

4-4-2-1 BmpCnv

BmpCnvはコマンドラインのツールです。makeファイルやバッチファイルに組み込んで使うことが前提になっています。いちど設定すればとくに意識しなくても使えるよう(自動的)になっています。make1.batに具体的な記述があります。

グラフィックデータは.¥bitmapsに格納されています。出力結果は同じディレクトリに出力されます。たとえば…。

BmpCnv .¥bitmaps¥gameBoard.bmp

となっている場合は

.¥bitmaps¥gameBoard.c

が出力されます。ReversiGame.cの冒頭のインクルードは、これらのグラフィックファイルを指しています。

グラフィックファイルの中身は、このようになります。

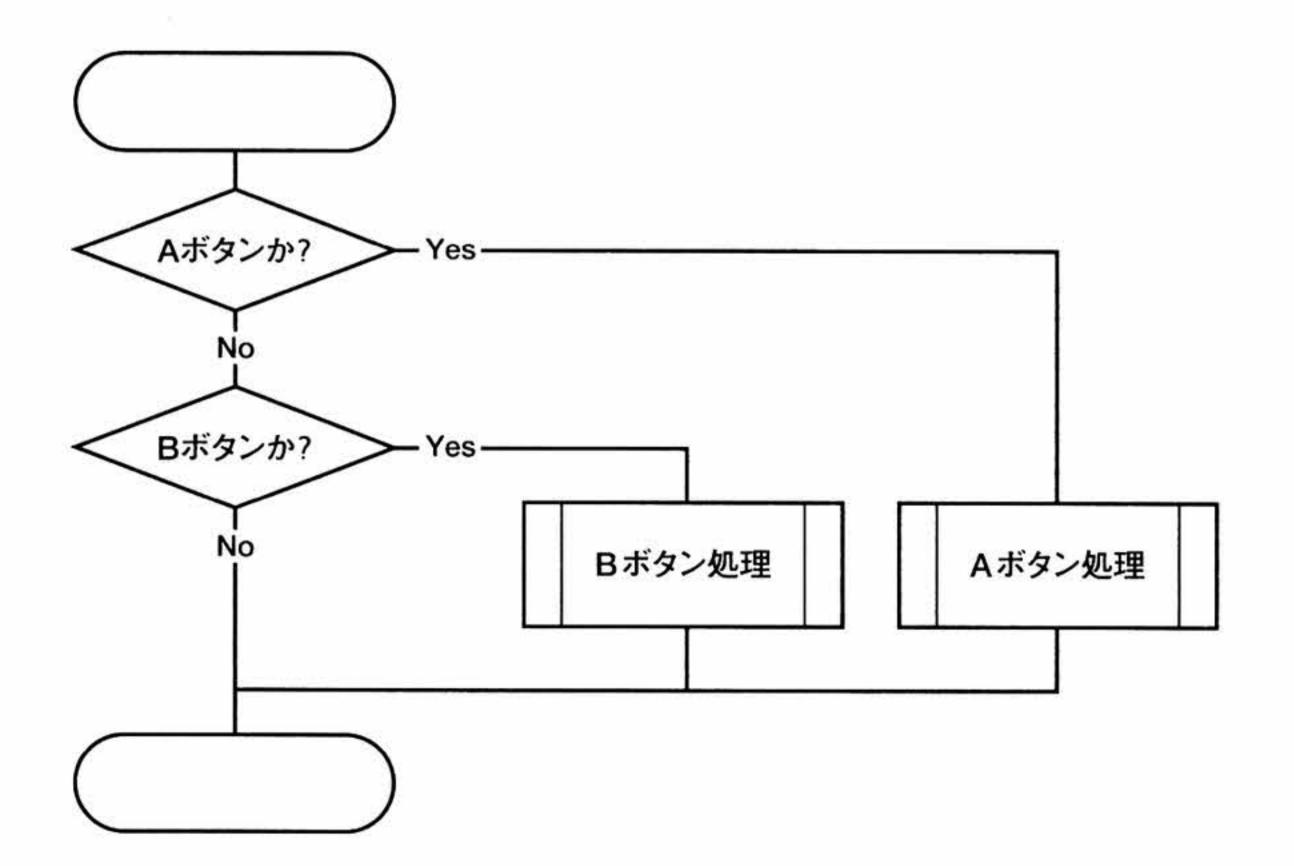
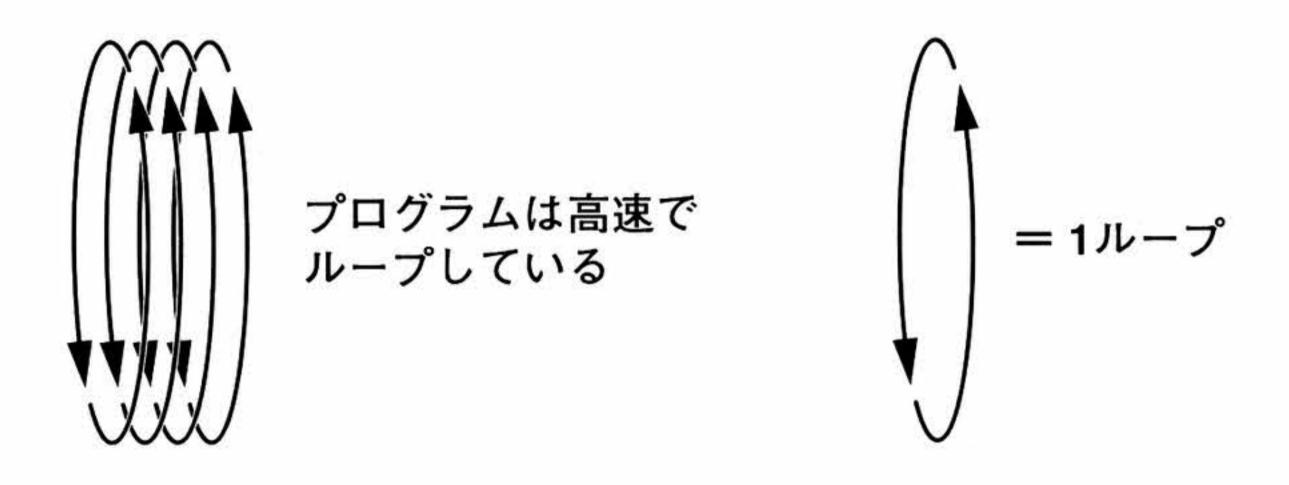
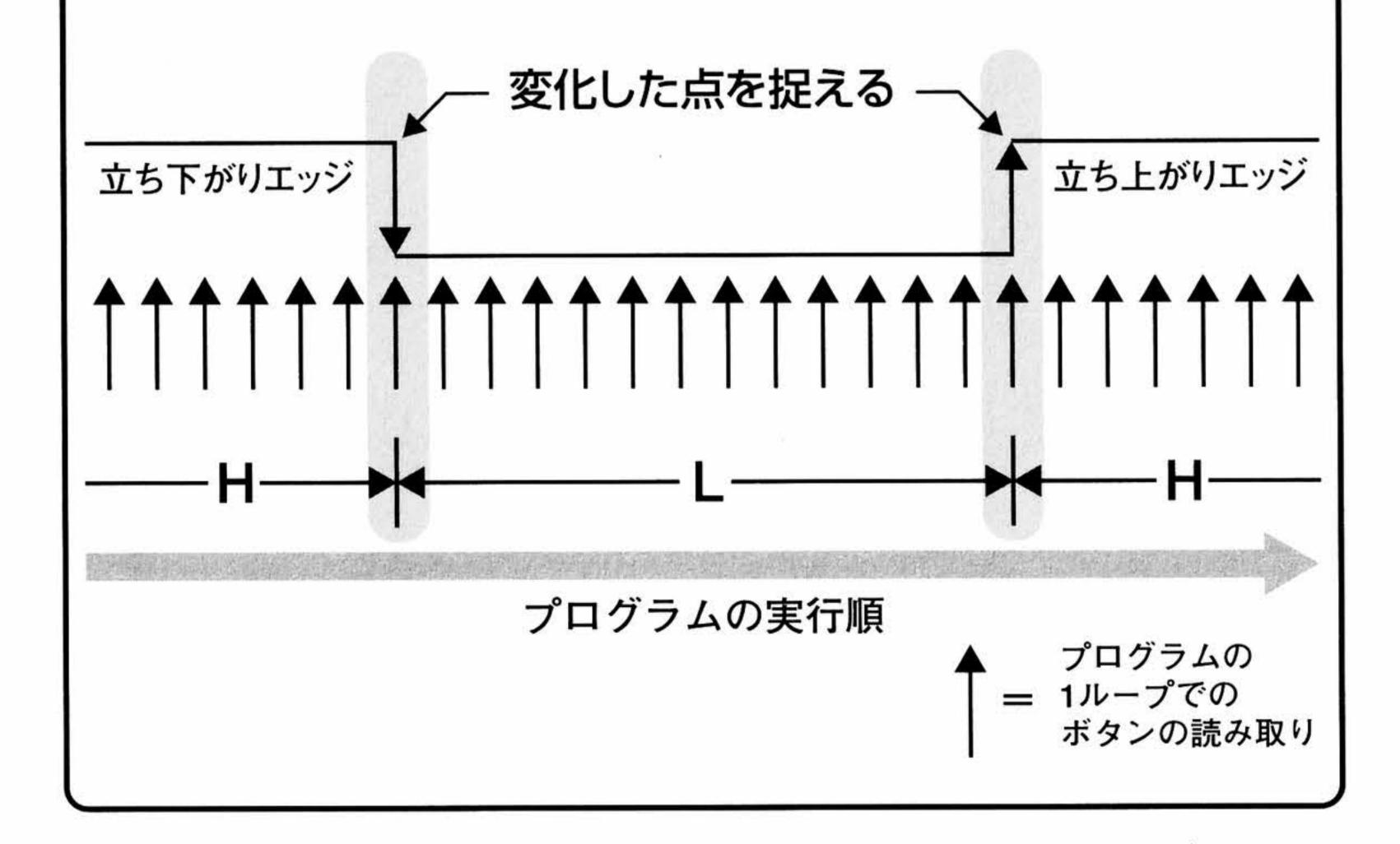
```
const u16 gameBoard_Map[] = {
0x f f f f, 0xe318, 0xe318, 0x f f 15, 0xe318, 0xe318,
0x050a, 0x8000, 0xe318, 0x0702, 0x8000, 0xe318,
……
```

ここで、ファイル名_Map[] が変数名となります。

今回のプログラムのように人工的な画面の場合では、オリジナルの容量に比べて5%～10%くらいの容量にまで圧縮されます。しかし、写真のような自然画のデータの場合はほとんど圧縮が効きません。用途に応じて使い分ける必要があります。

Column

1つ前のボタンの情報が必要なわけは？

ボタンの状態を見て、プログラムの分岐を行なう場合は、下のようなプログラムを作れば動作しそうな気がします。ところがこれではうまくいきません。なぜでしょうか？

なぜなら、プログラムの実行は人間の感覚でいえば非常に高速なので、ボタンの値を何度も読み取ってしまうからです。これを防ぐためにはボタンの変化(エッジ)を認識するようにします。エッジを認識するには1つ前の状態を把握していなければいけないので「1つ前のボタンの情報が必要」になってくるわけです。

ボタンのレベルだけで入力の判断を行なうと
複数回の入力があったように見える

makefile と make1.bat の作成

プログラムをビルドするときは、コマンドプロンプトから make1.bat を実行することは先に述べました。今回作成する make1.bat で行なっていることは、プログラムで使用されているビットマップを圧縮することと make プログラムを起動するのが主な働きになります。新たにビットマップファイルを使用したいときは、BmpCnv の行を付け加えるようにしてください。

BONSAI-WARE の make1.bat

```
001:  BmpCnv .¥bitmaps¥gameBoard.bmp
002:  BmpCnv .¥bitmaps¥ColorSelection.bmp
003   rem BmpCnv .¥bitmaps¥FingerCursorReady.bmp
      .¥bitmaps¥FingerCursorReady_mask.bmp
004:  BmpCnv .¥bitmaps¥LevelSelection.bmp
005:  BmpCnv .¥bitmaps¥opening.bmp
006:
007:  BmpCnv .¥bitmaps¥Stone_Black_16x16.bmp
      .¥bitmaps¥Stone_16x16_mask.bmp
008:  BmpCnv .¥bitmaps¥Stone_White_16x16.bmp
      .¥bitmaps¥Stone_16x16_mask.bmp
009:
010:  rem make -f makefile.txt
011:  make
012:
013:  pause
```

makeはMakefileを呼び出します。makeは効率良くプログラムをビルドするためにプログラムの依存関係を調べて、必要なファイルのみをコンパイルしてリンクするためのシステムです。makeについて細かく説明しようとすると本をまるまる一冊費やしても全部カバーすることができないくらいです(実際に数冊出版されており、筆者も持っています)。ここでは自分でプログラムを書く場合に必要な部分のみを解説します。

BONSAI-WAREのmakefile

```
001: #----------------------------------------------------------------------------------
002: # GBACC makefile
003  #----------------------------------------------------------------------------------
004:
005: #
006: # You have to re-write for your target.
007: #
008: NAME = ReversiAdv
009: .CFILES = ReversiMain.c ReversiOpening.c ReversiColorSelect.c
     ReversiLevelSelect.c ReversiGame.c ReversiEngine.c
010: .SFILES= crt0.S
011: #
012: #
013: #
014:
015: .OFILES = $(.CFILES:.c=.o)
016:
017: MAPFILE = $(NAME).map
018: TARGET_ELF = $(NAME).elf
019: TARGET_BIN = $(NAME).mb.gba
020:
021: .PHONY: all
022: all: $(TARGET_BIN)
023:
024: CFLAGS = -O3 -Wall -mthumb -mthumb-interwork
```

```
025:  ASFLAGS = -mthumb-interwork
026:  LDFLAGS = -mthumb-interwork
027:  LD = gcc
028:
029:  $(TARGET_BIN) : $(TARGET_ELF)
030:   objcopy -O binary $< $@
031:
032:  $(TARGET_ELF): crt0.o $(.OFILES)
033:   $(LD) $(LDFLAGS) $(.OFILES) -o $@
034:
035:  %.o: %.c
036:   gcc $(CFLAGS) -c $< -o $@
037:
038:  %.o: %.S
039:   as $(ASFLAGS) $< -o $@
040:
041:  #-------------------------------------------------------------------------------
042:  # end
043:  #-------------------------------------------------------------------------------
```

の部分がプログラムによって書き換える必要があるところです。NAME = で定義される部分は、どのような名前を実行ファイルに付けるのかを決めます。今回はReversiAdvになっているのがわかります。

.CFILES =にはビルドに必要なCのソースファイルをすべて記述します。

最後の.SFILES =はエントリを記述したアセンブラのソースリストです。

基本的には先のmake1.batファイルほど書き換える必要はありません。自分独自のエントリを書いた場合のみ書き換えます。

ReversiAdvanceのエンジン

最後に、このReversiAdvanceのエンジンに当たるプログラムを掲載しておきましょう。エンジンにはリバーシのルールや実際のプレイに伴う思考ルーチンがプログラムされています。このエンジン部分はGBAにあまり依存しないプログラムのため、詳しい解説は割愛しますが、どのようにしてReversiAdvanceが動作しているのか、プログラムから読み取ってほしいと思います。

ReversiEngine.h

```
001:  // ReversiEngine.h
002:
003   #ifndef REVERSI_ENGINE
004:  #define REVERSI_ENGINE
005:
006:  #define ID_STONE_NONE        0
007:  #define ID_STONE_BLACK       1
008:  #define ID_STONE_WHITE       2
009:
010:  #define GAME_STATUS_NORMAL          0
011:  #define GAME_STATUS_END             1
012:  #define GAME_STATUS_BLACK_PASS      2
013:  #define GAME_STATUS_WHITE_PASS      3
014:  #define GAME_STATUS_BLACK_PUT       4
015:  #define GAME_STATUS_WHITE_PUT       5
016:
017:  #define GAME_TURN_WHITE      1
018:  #define GAME_TURN_BLACK      0
019:
020:  #endif
```

ReversiEngine.c

```
001:  #include "gba.h"
002:
003:  // ReversiAdvance Engine part
004:  #include "ReversiConstants.h"
005:
006:  #include "ReversiEngine.h"
007:
008:  extern u16 gGameLevel;
009:
010:  u16 brdBaseInfo[BOARD_GRID_NUMBER][BOARD_GRID_NUMBER];
011:  u16 gGameStatusChanged;
012:
013:  void InitBaseInfo()
014:  {
015:     u16 i, j;
016:
017:     for(j = 0;j < BOARD_GRID_NUMBER;j++){
018:       for(i = 0;i < BOARD_GRID_NUMBER;i++){
019:         brdBaseInfo[i][j] = ID_STONE_NONE;
020:       }
021:     }
022:
023:     brdBaseInfo[3][3] = ID_STONE_WHITE;
024:     brdBaseInfo[3][4] = ID_STONE_BLACK;
025:     brdBaseInfo[4][3] = ID_STONE_BLACK;
026:     brdBaseInfo[4][4] = ID_STONE_WHITE;
027:
028:  }
029:
030:  u16 GetStone(u16* brd, u16 aStonePosX, u16 aStonePosY)
031:  {
032:     return (u16)*(brd + aStonePosY + aStonePosX * BOARD_GRID_NUMBER);
033:  }
034:
035:  u16 CheckPosition(u16* brd, u16 aStonePosX, u16 aStonePosY, u16
      aMyColor, u16 aWithReverse)
```

```
036:     {
037:     s16 targetColor;
038:     s16 aroundNumber, dX, dY;
039:     s16 aroundX, aroundY;
040:     s16 conditionScopeX, conditionScopeY;
041:     s16 linedColorX, linedColorY;
042:
043      // Get Target Color
044:     targetColor = ID_STONE_WHITE;
045:     switch (aMyColor)
046:       {
047:       case ID_STONE_NONE:
048:         break;
049:       case ID_STONE_BLACK:
050:         targetColor = ID_STONE_WHITE;
051:         break;
052:       case ID_STONE_WHITE:
053:         targetColor = ID_STONE_BLACK;
054:         break;
055:       default:
056:         targetColor = ID_STONE_WHITE;
057:         break;
058:       }
059:
060:     aroundNumber = 0;
061:     // Check correct position and reverse
062:     if(GetStone(brd,aStonePosX,aStonePosY) == ID_STONE_NONE){
063:       for(dY = -1;dY <= 1;dY++){
064:         for(dX = -1;dX <= 1;dX++){
065:           if((dX == 0) && (dY == 0)){
066:             // Point self
067:           }
068:           else{
069:             aroundX = aStonePosX + dX;
070:             aroundY = aStonePosY + dY;
071:             conditionScopeX = ((aroundX >= 0) && (aroundX < BOARD_
  GRID_NUMBER));
072:             conditionScopeY = ((aroundY >= 0) && (aroundY < BOARD_
```

```
        GRID_NUMBER));
073:
074:            if(conditionScopeX && conditionScopeY){
075:              if(GetStone(brd,aroundX,aroundY) == targetColor){
076:                linedColorX = aroundX;
077:                linedColorY = aroundY;
078:                while((linedColorX >= 0) && (linedColorX < BOARD_GRID_
        NUMBER) &&
079:                  (linedColorY >= 0) && (linedColorY < BOARD_GRID_
        NUMBER) &&
080:                  (GetStone(brd, linedColorX, linedColorY) == targetColor)){
081:                  linedColorX = linedColorX + dX;
082:                  linedColorY = linedColorY + dY;
083:                }
084:
085:                if((linedColorX >= 0) && (linedColorX < BOARD_GRID_
        NUMBER) &&
086:                  (linedColorY >= 0) && (linedColorY < BOARD_GRID_
        NUMBER) &&
087:                  GetStone(brd, linedColorX, linedColorY) == aMyColor){
088:
089:                  if(aWithReverse){
090:                    linedColorX = aroundX;
091:                    linedColorY = aroundY;
092:                    while(GetStone(brd, linedColorX, linedColorY) == targetColor){
093:                      // GetStone(brd, linedColorX, linedColorY) = aMyColor;
094:                      (u16)*(brd + linedColorY + linedColorX * BOARD_
        GRID_NUMBER) = aMyColor;
095:                      linedColorX = linedColorX + dX;
096:                      linedColorY = linedColorY + dY;
097:                    }
098:                    // GetStone(brd, aStonePosX, aStonePosY) = aMyColor;
099:                    (u16)*(brd + aStonePosY + aStonePosX * BOARD_
        GRID_NUMBER) = aMyColor;
100:                  }
101:                  aroundNumber++;
102:                }
103:              }
```

```
104:                 }
105:             }
106:         }
107:     }
108:   }
109:   return aroundNumber;
110: }
111:
112: u16 GameStatusCheck(u16 aNextColor)
113: {
114:   // Results ...
115:   // (0) Playing ... Possible to put stone
116:   // (1) No put position ... GameEnd
117:   // (2) Pass Black
118:   // (3) Pass White
119:   // (4) Possible to put Black
120:   // (5) Possible to put White
121:
122:   u16 i, j, result, blackStatus, whiteStatus;
123:
124:   blackStatus = whiteStatus = 0;
125:
126:   result = GAME_STATUS_NORMAL;
127:
128:   for(j = 0;j < BOARD_GRID_NUMBER;j++){
129:     for(i = 0;i < BOARD_GRID_NUMBER;i++){
130:       blackStatus = blackStatus + CheckPosition((u16*)brdBaseInfo, i, j,
     ID_STONE_BLACK, 0);
131:       whiteStatus = whiteStatus + CheckPosition((u16*)brdBaseInfo, i, j,
     ID_STONE_WHITE, 0);
132:     }
133   }
134:
135:   if(blackStatus + whiteStatus){
136:     if (aNextColor == ID_STONE_BLACK && blackStatus == 0)
137:       return GAME_STATUS_BLACK_PASS;
138:     if (aNextColor == ID_STONE_WHITE && whiteStatus == 0)
139:       return GAME_STATUS_WHITE_PASS;
```

```
140:     }
141:     else{
142:       return GAME_STATUS_END;
143:     }
144:
145:     return result;
146: }
147:
148: // ==============================================================
149: //
150: //
151: //
152: // ==============================================================
153:
154: struct Evaluation
155: {
156:     u16 x;
157:     u16 y;
158:     s16 score;
159: };
160:
161: struct Evaluation ReadNextMove(u16* brd, u16 readMoves, u16
     aComColor, u16 aPutColor);
162: struct Evaluation GetEvaluate(u16* brd, u16 aComColor);
163: u16 CornerEvaluate(u16* board, u16 aPutColor);
164:
165: void CopyBoard(u16* newbrd, u16* orgbrd)
166: {
167:     u16 i, j;
168:     for(j = 0;j < BOARD_GRID_NUMBER;j++){
169:       for(i = 0;i < BOARD_GRID_NUMBER;i++){
170:         (u16)*(newbrd + i + j * BOARD_GRID_NUMBER) = (u16)*(orgbrd + i
     + j * BOARD_GRID_NUMBER);
171:       }
172:     }
173: }
174:
175: u16 ReversiEngine(u16 aComColor, u16 aComLevel)
```

```
176: {
177:     u16 x, y, i, j;
178:     struct Evaluation eva;
179:     u16 myBrd[BOARD_GRID_NUMBER][BOARD_GRID_NUMBER];
180:
181:     x = 0;
182:     y = 0;
183:
184:     CopyBoard(&(myBrd[0][0]), &(brdBaseInfo[0][0]));
185:     if (aComLevel == 0)
186:     {
187:       for(j = 0;j < BOARD_GRID_NUMBER;j++){
188:         for(i = 0;i < BOARD_GRID_NUMBER;i++){
189:           if (CheckPosition(&(brdBaseInfo[0][0]), i, j, aComColor, 1))
190:             return 0;
191:         }
192:       }
193:     }
194:     eva = ReadNextMove(&(myBrd[0][0]), aComLevel, aComColor, aComColor);
195:     CheckPosition(&(brdBaseInfo[0][0]), eva.x, eva.y, aComColor, 1);
196:
197:     return 0;
198:
199: }
200:
201: u16 PutAbleCount(u16* brd, u16 aComColor)
202: {
203:     u16 i, j, count;
204:     count = 0;
205:     for(j = 0;j < BOARD_GRID_NUMBER;j++){
206:       for(i = 0;i < BOARD_GRID_NUMBER;i++){
207:         if(CheckPosition(brd, i, j, aComColor, 0)){
208:           count++;
209:         }
210:       }
211:     }
212:     return count;
213: }
```

```
214:
215: u16 PieceCount(u16* brd, u16 aComColor)
216: {
217:    u16 i, j, count;
218:    count = 0;
219:    for(j = 0;j < BOARD_GRID_NUMBER;j++){
220:      for(i = 0;i < BOARD_GRID_NUMBER;i++){
221:        if(GetStone(brd,i, j) == aComColor){
222:          count++;
223:        }
224:      }
225:    }
226:    return count;
227: }
228:
229: struct Evaluation ReadNextMove(u16* brd, u16 readMoves, u16
     aComColor, u16 aPutColor)
230: {
231:    u16 i, j;
232:    u16 myBrd[BOARD_GRID_NUMBER][BOARD_GRID_NUMBER];
233:
234:    struct Evaluation best;
235:    struct Evaluation worst;
236:    struct Evaluation eva;
237:
238:    best.score = -9999;
239:    worst.score = 9999;
240:    if (readMoves == 0)
241:    {
242:      return GetEvaluate(brd, aComColor);
243:    }
244:    if (PutAbleCount(brd, aPutColor) == 0)
245:    {
246:      if (PutAbleCount(brd, (3 - aPutColor)) == 0)
247:      {
248:        if (PieceCount(brd, aComColor) > PieceCount(brd, (3 - aComColor)))
249:        {
250:          eva.score = 7000 + PieceCount(brd, aComColor);
```

```
251:             return eva;
252:           } else {
253:             eva.score = - 7000 + PieceCount(brd, aComColor);;
254:             return eva;
255:           }
256:         }
257:         eva = ReadNextMove(&(myBrd[0][0]), readMoves - 1, aComColor, (3 - 
     aPutColor));
258:         return eva;
259:       }
260: 
261:       CopyBoard(&(myBrd[0][0]), brd);
262:       for(j = 0;j < BOARD_GRID_NUMBER;j++){
263:         for(i = 0;i < BOARD_GRID_NUMBER;i++){
264:           if(CheckPosition(&(myBrd[0][0]), i, j, aPutColor, 1)){
265:             eva = ReadNextMove(&(myBrd[0][0]), readMoves - 1, aComColor, 
     (3 - aPutColor));
266:             if (eva.score < worst.score)
267:             {
268:               worst.score = eva.score;
269:               worst.x = i;
270:               worst.y = j;
271:             }
272:             if (eva.score > best.score)
273:             {
274:               best.score = eva.score;
275:               best.x = i;
276:               best.y = j;
277:             }
278:             CopyBoard(&(myBrd[0][0]), brd);
279:           }
280:         }
281:       }
282:       if (aComColor == aPutColor) {
283:         return best;
284:       } else {
285:         return worst;
286:       }
```

```
287: }
288:
289: struct Evaluation GetEvaluate(u16* brd, u16 aComColor)
290: {
291:     s16 i, j, score;
292:     s16 dY, dX;
293:     struct Evaluation eva;
294:     score = 0;
295:
296:     for(j = 0;j < BOARD_GRID_NUMBER;j++){
297:       for(i = 0;i < BOARD_GRID_NUMBER;i++){
298:         if (GetStone(brd, i, j) == ID_STONE_NONE) {
299:           for(dY = -1;dY <= 1;dY++){
300:             for(dX = -1;dX <= 1;dX++){
301:               if((dX == 0) && (dY == 0)){
302:                 // Point self
303:               }
304:               else
305:               {
306:                 if ((i+dX) >= 0 && (i+dX) <= 7 && (j+dY) >= 0 && (j+dY) <= 7)
307:                 {
308:                   if (GetStone(brd,i + dX,j + dY) == aComColor) {
309:                     score--;
310:                   }
311:                   if (GetStone(brd,i + dX,j + dY) == (3 - aComColor)) {
312:                     score++;
313:                   }
314:                 }
315:               }
316:             }
317:           }
318:         }
319:       }
320:     }
321:
322:     eva.score = score + CornerEvaluate(brd, aComColor);
323:     return eva;
324: }
```

```
325:
326: u16 CornerEvaluate(u16* board, u16 aPutColor)
327: {
328:    u16 score = 0;
329:    if (GetStone(board, 1, 1) == (3-aPutColor)) {
330:      if (GetStone(board, 0, 0) == ID_STONE_NONE) {
331:        score = score + 120;
332:      }
333:    }
334:    if (GetStone(board, 6, 1) == (3-aPutColor)) {
335:      if (GetStone(board, 7,0) == ID_STONE_NONE) {
336:        score = score + 120;
337:      }
338:    }
339:    if (GetStone(board, 1, 7) == (3-aPutColor)) {
340:      if (GetStone(board, 0, 7) == ID_STONE_NONE) {
341:        score = score + 120;
342:      }
343:    }
344:    if (GetStone(board, 6, 6) == (3-aPutColor)) {
345:      if (GetStone(board, 7, 7) == ID_STONE_NONE) {
346:        score = score + 120;
347:      }
348:    }
349:
350:    if (GetStone(board, 1, 1) == aPutColor) {
351:      if (GetStone(board, 0, 0) == ID_STONE_NONE) {
352:        score = score - 120;
353:      }
354:    }
355:    if (GetStone(board, 6, 1) == aPutColor) {
356:      if (GetStone(board, 7, 0) == ID_STONE_NONE) {
357:        score = score - 120;
358:      }
359:    }
360:    if (GetStone(board, 1, 6) == aPutColor) {
361:      if (GetStone(board, 0, 7) == ID_STONE_NONE) {
362:        score = score - 120;
```

```
363:        }
364:    }
365:
366:    if (GetStone(board, 6, 6) == aPutColor) {
367:        if (GetStone(board, 7, 7) == ID_STONE_NONE) {
368:            score = score - 120;
369:        }
370:    }
371:
372:    if (GetStone(board, 0, 0) == aPutColor) score = score + 200;
373:    if (GetStone(board, 7, 0) == aPutColor) score = score + 200;
374:    if (GetStone(board, 0, 7) == aPutColor) score = score + 200;
375:    if (GetStone(board, 7, 7) == aPutColor) score = score + 200;
376:
377:    if (GetStone(board, 0, 0) == (3-aPutColor)) score = score - 200;
378:    if (GetStone(board, 7, 0) == (3-aPutColor)) score = score - 200;
379:    if (GetStone(board, 0, 8) == (3-aPutColor)) score = score - 200;
380:    if (GetStone(board, 7, 7) == (3-aPutColor)) score = score - 200;
381:
382:
383:    if(CheckPosition(board, 0, 0, aPutColor, 0)) score = score + 80;
384:    if(CheckPosition(board, 7, 0, aPutColor, 0)) score = score + 80;
385:    if(CheckPosition(board, 0, 7, aPutColor, 0)) score = score + 80;
386:    if(CheckPosition(board, 7, 7, aPutColor, 0)) score = score + 80;
387:
388:    if(CheckPosition(board, 0, 0, (3-aPutColor), 0)) score = score - 80;
389:    if(CheckPosition(board, 7, 0, (3-aPutColor), 0)) score = score - 80;
390:    if(CheckPosition(board, 0, 7, (3-aPutColor), 0)) score = score - 80;
391:    if(CheckPosition(board, 7, 7, (3-aPutColor), 0)) score = score - 80;
392:
393:    return score;
394: }
```

Column

ヘッダーファイルを使うわけ

我々が日常で数を使う場合は、数学の問題を解くような場合を除いて、何らかの実体と結びついているのが普通です。「5 個のりんご」とか「3 本の鉛筆」とか「1 だから OK」「0 だから NG」などのようにです。プログラムも同様で、 **a = 1;** と記述した場合は、a に 1 を代入するというだけでなく、その 1 がいったい何を意味しているのか、実体と結びついているほうがわかりやすいわけです。ところが、プログラムを記述した当人以外にはわからないというプログラムが多いのも事実です。

しかし、第三者が見る機会の多いオープンソースの場合、プログラムを記述した当人にしかわからないのでは困ります。そうした事態を防ぐために、これまではコメントをたくさんプログラム中に記述したわけですが、これはこれでプログラムが煩雑で読みにくくなるという欠点や難解なコメントでは意味をなさないという問題が生じます。そこであらかじめ、その数字が意味する実体をプログラム中に記述し、数値は定数(じょうすう)としてヘッダファイル(*.h)に定義しておくわけです。

```
#define FIRST_NUMBER 1
```

たとえば、上のようにヘッダファイルに定数を定義してプログラムを

```
a = FIRST_NUMBER;
```

このようにすると、**a は最初の数として初期化された**ことがわかります。冗長なコメントが不要になるほど、雄弁な(?)ソースコードが書けるわけです。FIRST_NUMBER が実体で、1 がその定数というわけです。上記の例は「マジックナンバー」と言われていて、ソースコードを記述するときのマナーでもあります。憶えておいてください。

自作ライブラリの説明

TeamKNOxLib

“リバーシ”で使用したTeamKNOxオリジナルライブラリについて解説します。このライブラリはソースのまま使用しているので、自分で作成した関数を簡単に登録できますし、ライブラリをどのように作っていくのかも学ぶことができるでしょう。

GBAプログラミングでは独自のライブラリ構築が不可欠！

ここではGBAプログラミングを通して制作した関数をまとめて説明したいと思います。これらをまとめたものをライブラリと呼びます。GBAに限らず、ソフトウェアを作っていくと、ある程度のパターンができてきます。とくにC言語以降の言語は、関数を定義していくことでそれ自身を拡張していくことが可能です。PCのC言語プログラミングではstdio.hやstdlib.hなどがおなじみですが、これらをまともに実装するとGBAにとっては大変サイズの大きなものになってしまいます。そこで必要最小限のライブラリを独自に構築する必要があります。

これらのライブラリはTeamKNOxで共通して利用しているものです。まだまだ発展途上（とくに音関係）なので強化していく必要はありますが、現状でも今回紹介したリバーシ程度のものは十分に組むことが可能です。

ライブラリの使い方については、実際のプログラム（BONSAI-WARE）を参照していただくとより理解が深まることと思います。

ライブラリの分類と使い方

ライブラリはそのプログラムの目的により、いくつかに分類することができます。現状では大きく分けて以下のようになります。

グラフィックライブラリ

グラフィックは最も基本的なものです。VRAMの操作を簡便に行なうためにいくつかの基本的な関数とその関数を内包する関数で構成されています。現在のところ以下の関数が用意されています。

関数	void SetPixel(u16 aX, u16 aY, u16 aColor, u32 aVRAM);
説明	指定したVRAMに指定した色で点を描画します

関数	void DrawLine(u16 aX0, u16 aY0, u16 aX1, u16 aY1, u16 aColor, u32 aVRAM);
説明	指定したメモリ領域に指定した色で線を描画します

関数	void DrawCircle(u16 aX, u16 aY, u16 aRadius, u16 aColor, u32 aVRAM);
説明	指定したメモリ領域に指定した色で円を描画します

関数	void DrawBox(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16 aColor, u32 aVRAM);
説明	指定したメモリ領域に指定した色で四角形を描画します(塗りつぶし)

関数	void DrawBoxHalf(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u32 aVRAM);
説明	指定したメモリ領域に半透明の四角形を描画します

関数	void BitBltSRCCOPY(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u32 aVRAM);
説明	指定したメモリ領域にイメージを転送します

関数	void BitBltOR(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u32 aVRAM);
説明	指定したメモリ領域にイメージをORの演算で転送・描画します

関数	void BitBltAND(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u32 aVRAM);
説明	指定したメモリ領域にイメージをANDの演算で転送・描画します

関数	void BitBltExOR(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u32 aVRAM);
説明	指定したメモリ領域にイメージをEX-ORの演算で転送・描画します

関数	void BitBltMasked(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u16* aMaskImage, u32 aVRAM);
説明	指定したメモリ領域にイメージをマスクして転送・描画します

関数	void BitBltMaskedComp(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u32 aVRAM);
説明	指定したメモリ領域にBmpCnvの圧縮イメージを転送・描画します

関数	void DrawTextInit();
説明	DrawText() を初期化します

関数	void DrawText(u16 aX, u16 aY, s8* aStrings, u16 aStringColor, u16 aBGColor, u16 aRectMask, u32 aVRAM);
説明	文字列を描画します

関数	void Off2VRAM(u32 aSourceAddress, u32 aDistAddress);
説明	メモリの転送を行ないます。メモリの特定領域を仮想的なVRAMとして確保してそれを実VRAMへ転送するのに用います

関数	void DrawBoxEmpty(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16 aColor, u32 aVRAM);
説明	指定したメモリ領域に指定した色で四角形を描画します(塗りつぶさない)

関数	void WaitForVsync(void);
説明	垂直帰線同期を待ちます

通信ライブラリ

GBAになってこれまでのGBCと大きく変わったのが、通信機能の強化です。PCなどで多く用いられている調歩同期式の通信(EIA-232Cなどですね)もサポートされるようになりました。これにより、さまざまな周辺機器を接続することが可能になりました。

関数	void InitUART(u16 parameter);
説明	通信ポートの初期化を行ないます。パラメータは以下のとおりです

パラメータ	BAUD_RATE_9600　0x0000 BAUD_RATE_38400　0x0001 BAUD_RATE_57600　0x0002 BAUD_RATE_115200　0x0003 CTS_ENABLE　0x0004 SEND_DATA_FULL　0x0010 RECV_DATA_EMPTY　0x0020 ERROR　0x0040 DATA_LENGTH_7　0x0000 DATA_LENGTH_8　0x0080 FIFO_ENABLE　0x0100 PARITY_NONE　0x0000 PARITY_EVEN　0x0200 PARITY_ODD　0x0208 SEND_ENABLE　0x0400 RECV_ENABLE　0x0800 UART_MODE　0x3000 SIO_IRQ_ENABLE　0x4000 ●これらをORで組み合わせてパラメータをセットします。

関数	void SendByte(u8 data);
説明	1バイトデータを通信ラインに乗せます

文字列操作ライブラリ

簡単な文字列操作が行なえるようになっています。PC関連ではstring.hなどのライブラリがおなじみですが、現在では数値から文字列への変換のみが実装されています。

関数	void num02str(u8 number);
説明	2桁の数(8ビット分)を2char分の文字列に変換します

関数	void num04str(u16 number);
説明	4 桁の数(16ビット分)を4char 分の文字列に変換します

乱数ライブラリ

ゲームを行なう上で必要な乱数を生成するための関数群です。この辺はゲームの味付け(ゲーム性)と直結しているため、商用ゲームではノウハウの塊です。このライブラリには、最も簡単なものを実装しています。必要に応じて組み分けるといいでしょう。

関数	void seed_lc(u32 v);
説明	乱数の種をセットします。乱数の種にはキー入力時の待ち時間などを利用したり、センサーなどの入力値を利用します。なるべく、不定な値になるようにしてください

関数	u32 random_lc(void);
説明	Linear Congruential (LC) - 線形合同法による乱数の発生をします

ボタン操作ライブラリ

ボタンの操作を行ないます。

関数	void AWait();
説明	Aボタンが押されるまで待ちます

時間ライブラリ

時間を取り扱う関数です。

関数	void Wait(u32 aWaitingTime);
説明	aWaitingTimeで指定した分だけループして待ちます

その他のライブラリ

GBA特有の関数です。PCなどではお目にかかれない機能を実装してあります。

関数	void GetTitle(s8 *aGameTitle);
説明	ROMに格納されている商用カートリッジのゲームタイトルを取得します

グローバル変数

ライブラリを構成する上で必要なグローバル変数です。

関数	s8 WorkStr[8];
説明	文字列操作の結果の格納用です

関数	u32 gRandomSeed;
説明	乱数の種を保持します

ソースコード

5-3-1 TeamKNOxLibの仕組み

これらのライブラリはすべて、ソースコードをそのつどコンパイルしています。通常、ライブラリアンというコンパイラに付属するツールで*.libファイルの形式にしてリンク時に必要なライブラリのみをリンクするのが常道です。ただ、TeamKNOxLibそのものはまだまだ発展途上なので、ソースコードそのものをコンパイルするかたちにしました。現在のPCは十分に高速ですから、その都度コンパイルしてもコンパイル待ちでイライラすることはありません。また、ソースコードを読むことはプログラミングのスキルをあげる上で大変重要です。

5-3-2 TeamKNOxLibのこれから…

気をつけてプログラミングしていますが、バグなどが潜在しているかもしれません。問題が見つかったらフィードバックしていただけると幸いです。また、グラフィックはモード3のみ実装していますが、GBAにはさまざまなグラフィックモードが存在します。今後は、それらと音関係を充実させていきたいと考えています。

TeamKNOxLib.h

```
001:  // TeamKNOxLib.h
002:
003:  #ifndef TEAMKNOX_LIB
004:  #define TEAMKNOX_LIB
005:
006:  #include "bkg.c"
007:  #include "SIO.h"
008:
009:  s8 gWorkStr[8];
010:
011:  void num02str( u8 number )
012:  {
013:      u16 m;
014:      u8 i, n, f;
015:      char work[4];
016:
017:      f = 0; m = 10;
018:      for(  i=0; i<2; i++  ) {
019:          n = number / m; number = number % m; m = m / 10;
020:          if(  ( n==0 )&&( f==0 )  ) {
021:  //              work[i] = ' ';          // ' '
022:                  work[i] = '0';          // '0'
023:          } else {
024:                  f = 1;
025:                  work[i] = n + '0';   //'0' - '9'
026:          }
027:      }
028:      work[i] = 0x00;
029:
030:      i = n = 0;
031:      while( work[n] ){
032:          while( work[n] == ' ' ){
033:                  n++;
034:          }
```

```
035:            gWorkStr[i] = work[n];
036:            i++;
037:            n++;
038:        }
039:        gWorkStr[i] = 0x00;
040:    }
041:
042:    void num04str( u16 number )
043:    {
044:        u16 m;
045:        u8 i, n, f;
046:        char work[6];
047:
048:        f = 0; m = 1000;
049:        for( i = 0;i < 4;i++ ){
050:            n = number / m; number = number % m; m = m / 10;
051:            if(  ( n==0 )&&( f==0 )  ) {
052:    //              work[i] = ' ';          // ' '
053:                work[i] = '0';          // '0'
054:            } else {
055:                f = 1;
056:                work[i] = n + '0';   //'0' - '9'
057:            }
058:        }
059:        work[i] = 0x00;
060:
061:        i = n = 0;
062:        while( work[n] ){
063:            while( work[n] == ' ' ){
064:                n++;
065:            }
066:            gWorkStr[i] = work[n];
067:            i++;
068:            n++;
069:        }
070:        gWorkStr[i] = 0x00;
```

```
071:  }
072:
073:
074:  void WaitForVsync( void )
075:  {
076:      while ( *( volatile u16* )0x4000006 >= 160 ) {};
077:      while ( *( volatile u16* )0x4000006 < 160 ) {};
078:  }
079:
080:  void GetRGB( u8 *arRed, u8 *arGreen, u8 *arBlue, u16 aPosScreen, u32
      aVRAM )
081:  {
082:      u16* ScreenBuffer;
083:
084:      ScreenBuffer = ( u16* )aVRAM;
085:      *arRed = ScreenBuffer[aPosScreen] & 0x1F;
086:      *arGreen = ( ScreenBuffer[aPosScreen] & 0x03E0 ) >> 5;
087:      *arBlue = ( ScreenBuffer[aPosScreen] & 0x7C00 ) >> 10;
088:  }
089:
090:
091:  void SetPixel( u16 aX, u16 aY, u16 aColor, u32 aVRAM )
092:  {
093:      u16* ScreenBuffer;
094:
095:      ScreenBuffer = ( u16* )aVRAM;
096:      ScreenBuffer[aY * SCREEN_SIZE_X + aX] = aColor;
097:  }
098:
099:
100:  void DrawLine( u16 aX0, u16 aY0, u16 aX1, u16 aY1, u16 aColor, u32
      aVRAM )
101:  {
102:      s16 x, y = aY0;
103:      s16 dx = ( aX1 - aX0 ) * 2;
104:      s16 dy = ( aY1 - aY0 ) * 2;
```

```
105:
106:         s16 dydx = dy - dx;
107:         s16 d = dy - dx / 2;
108:
109:         u16* ScreenBuffer;
100:
111:         ScreenBuffer = ( u16* )aVRAM;
112:         for( x = aX0;x <= aX1; x++ ){
113:             ScreenBuffer[y * SCREEN_SIZE_X + x] = aColor;
114:             if( d < 0 ){
115:                 d = d + dy;
116:             }
117:             else{
118:                 y++;
119:                 d = d + dydx;
120:             }
121:         }
122:     }
123:
124:
125:     void DrawCircle( u16 aX, u16 aY, u16 aRadius, u16 aColor, u32 aVRAM )
126:     {
127:         s16 x, y, e;
128:         u16* ScreenBuffer;
129:
130:         ScreenBuffer = ( u16* )aVRAM;
131:
132:         x = aRadius;
133:         y = 0;
134:         e = 3 - 2 * aRadius;
135:
136:         while( y <= x ){
137:             ScreenBuffer[aX + x + ( aY + y ) * SCREEN_SIZE_X] = aColor;
138:             ScreenBuffer[aX + x + ( aY - y ) * SCREEN_SIZE_X] = aColor;
139:             ScreenBuffer[aX - x + ( aY + y ) * SCREEN_SIZE_X] = aColor;
140:             ScreenBuffer[aX - x + ( aY - y ) * SCREEN_SIZE_X] = aColor;
```

```
141:
142:            ScreenBuffer[aX + y + ( aY + x ) * SCREEN_SIZE_X] = aColor;
143:            ScreenBuffer[aX + y + ( aY - x ) * SCREEN_SIZE_X] = aColor;
144:            ScreenBuffer[aX - y + ( aY + x ) * SCREEN_SIZE_X] = aColor;
145:            ScreenBuffer[aX - y + ( aY - x ) * SCREEN_SIZE_X] = aColor;
146:
147:            if( e < 0 ){
148:                e = e + 4 * y + 6;
149:            }
150:            else{
151:                e = e + 4 * ( y - x ) + 10;
152:                x--;
153:            }
154:            y++;
155:        }
156:    }
157:
158:
159:    void DrawBox( u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16 aColor,
        u32 aVRAM )
160:    {
161:        u16 x, y;
162:        u16* ScreenBuffer;
163:
164:        ScreenBuffer = ( u16* )aVRAM;
165:        for( y = 0;y < aHeight;y++ ){
166:            for( x = 0;x < aWidth;x++ ){
167:                ScreenBuffer[( y + aY ) * SCREEN_SIZE_X + x + aX] = aColor;
168:            }
169:        }
170:    }
171:
172:    void DrawBoxEmpty( u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16
        aColor, u32 aVRAM )
173:    {
174:        DrawBox( aX, aY, aWidth + 2, 2, aColor, aVRAM );
```

```
175:        DrawBox( aX + aWidth, aY + 2, 2, aHeight, aColor, aVRAM );
176:        DrawBox( aX, aY + aHeight, aWidth, 2, aColor, aVRAM );
177:        DrawBox( aX, aY, 2, aHeight, aColor, aVRAM );
178:    }
179:
180:    void DrawBoxHalf( u16 aX, u16 aY, u16 aWidth, u16 aHeight, u32
        aVRAM )
181:    {
182:        u16 x, y, posScreen;
183:        u16* ScreenBuffer;
184:
185:        u8 red, green, blue;
186:
187:        ScreenBuffer = ( u16* )aVRAM;
188:        for( y = 0;y < aHeight;y++ ){
189:            for( x = 0;x < aWidth;x++ ){
190:                posScreen = ( y + aY ) * SCREEN_SIZE_X + x + aX;
191:                GetRGB( &red, &green, &blue, posScreen, aVRAM );
192:                red = red >> 1;
193:                green = green >> 1;
194:                blue = blue >> 1;
195:                ScreenBuffer[posScreen] = RGB( red, green, blue );
196:            }
197:        }
198:        DrawBoxEmpty( aX, aY, aWidth, aHeight, RGB( 20, 20, 20 ), aVRAM );
199:    }
200:
201:    void BitBltSRCCOPY( u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16*
        aImage, u32 aVRAM )
202:    {
203:        u16 x, y;
204:        u16* ScreenBuffer;
205:
206:        ScreenBuffer = ( u16* )aVRAM;
207:
208:        for( y = 0;y < aHeight;y++ ){
```

```
209:            for( x = 0;x < aWidth;x++ ){
210:                ScreenBuffer[( y + aY ) * SCREEN_SIZE_X + x + aX] =
    aImage[y * aWidth + x];
211:            }
212:        }
213:    }
214:
215:
216:    void BitBltOR( u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage,
    u32 aVRAM )
217:    {
218:        u16 x, y;
219:        u16* ScreenBuffer;
220:
221:        ScreenBuffer = ( u16* )aVRAM;
222:
223:        for( y = 0;y < aHeight;y++ ){
224:            for( x = 0;x < aWidth;x++ ){
225:                ScreenBuffer[( y + aY ) * SCREEN_SIZE_X + x + aX] =
    ScreenBuffer[( y + aY ) * SCREEN_SIZE_X + x + aX] ¦ aImage[y * aWidth
    + x];
226:            }
227:        }
228:    }
229:
230:
231:    void BitBltAND( u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage,
    u32 aVRAM )
232:    {
233:        u16 x, y;
234:        u16* ScreenBuffer;
235:
236:        ScreenBuffer = ( u16* )aVRAM;
237:
238:        for( y = 0;y < aHeight;y++ ){
239:            for( x = 0;x < aWidth;x++ ){
```

```
240:                 ScreenBuffer[( y + aY ) * SCREEN_SIZE_X + x + aX] =
     ScreenBuffer[( y + aY ) * SCREEN_SIZE_X + x + aX] & aImage[y *
     aWidth + x];
241:             }
242:         }
243: }
244:
245:
246: void BitBltExOR( u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16*
     aImage, u32 aVRAM )
247: {
248:     u16 x, y;
249:     u16* ScreenBuffer;
250:
251:     ScreenBuffer = ( u16* )aVRAM;
252:
253:     for( y = 0;y < aHeight;y++ ){
254:         for( x = 0;x < aWidth;x++ ){
255:             ScreenBuffer[( y + aY ) * SCREEN_SIZE_X + x + aX] =
     ScreenBuffer[( y + aY ) * SCREEN_SIZE_X + x + aX] ^ aImage[y *
     aWidth + x];
256:         }
257:     }
258: }
259:
260: void BitBltMasked( u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16*
     aImage, u16* aMaskImage, u32 aVRAM )
261: {
262:     u16 i, bitbltWork[32*32];
263:
264:     for( i = 0;i < aWidth * aHeight;i++ ){
265:         bitbltWork[i] = ~aMaskImage[i];
266:     }
267:
268:     BitBltOR( aX, aY, aWidth, aHeight, ( u16* )bitbltWork, aVRAM );
269:     BitBltAND( aX, aY, aWidth, aHeight, ( u16* )aImage, aVRAM );
```

```
270: }
271:
272: void BitBltMaskedComp( u16 aX, u16 aY, u16 aWidth, u16 aHeight,
     u16* aImage, u32 aVRAM )
273: {
274:     u16 x, y, i, size, rgb[2], a, j, k;
275:     u16* ScreenBuffer;
276:     u8 count[2];
277:     ScreenBuffer = ( u16* )aVRAM;
278:
279:     size = aWidth * aHeight;
280:     i = 0;
281:     a = 0;
282:     while( a < size ){
283:         count[0] = ( u8 )( aImage[i]>>8 );
284:         count[1] = ( u8 )( aImage[i]&0xFF );
285:         i++;
286:         for ( j = 0; j < 2; j++ ){
287:             rgb[j] = aImage[i];
288:             i++;
289:         }
290:         for ( k = 0; k < 2; k++ ){
291:             for ( j = 0; j < count[k];j++ ){
292:                 if  (( rgb[k]&0x8000 )!=0 ){
293:                     y = ( int )( a / aWidth + 1 );
294:                     x = a % aWidth;
295:                     ScreenBuffer[(( aHeight - y ) + aY ) * SCREEN_SIZE_X
     + x + aX] = ( rgb[k]&0x7FFF );
296:                 }
297:                 a++;
298:             }
299:         }
300:     }
301: }
302:
303: void DrawTextInit( )
```

```
304: {
305: }
306:
307: #define CHARCTER_SIZE 8
308: void DrawText( u16 aX, u16 aY, s8* aStrings, u16 aStringColor, u16
     aBGColor, u16 aRectMask, u32 aVRAM )
309: {
310:     int i, j, k;
311:     u16 CharBuffer[CHARCTER_SIZE*CHARCTER_SIZE],
     MaskBuffer[CHARCTER_SIZE*CHARCTER_SIZE], textPosX, temp;
312:     u8 charCode;
313:     u16 maskPattern[CHARCTER_SIZE*CHARCTER_SIZE];
314:
315:     for( i = 0;i < CHARCTER_SIZE*CHARCTER_SIZE;i++ ){
316:         maskPattern[i] = aBGColor;
317:     }
318:
319:
320:     textPosX = aX;
321:     while( *aStrings ){
322:         charCode = *aStrings;
323:
324:         k = 0;
325:         for( j = 0;j < CHARCTER_SIZE;j++ ){
326:             temp = bkg_Map[charCode * CHARCTER_SIZE + j];
327:
328:             for( i = 0;i < CHARCTER_SIZE;i++ ){
329:                 temp = temp << 1;
330:                 if ( temp & 0x0100 ){
331:                     CharBuffer[k] = aStringColor;
332:                     MaskBuffer[k] = 0x0000;
333:                 }
334:                 else{
335:                     CharBuffer[k] = aBGColor;
336:                     MaskBuffer[k] = aBGColor;
337:                 }
```

```
338:                k++;
339:            }
340:        }
341:
342:    if( aRectMask ){
343:        BitBltSRCCOPY( textPosX, aY, CHARCTER_SIZE,
CHARCTER_SIZE, ( u16* )maskPattern, aVRAM );
344:    }
345:        BitBltMasked( textPosX, aY, CHARCTER_SIZE,
CHARCTER_SIZE, ( u16* )CharBuffer, ( u16* )MaskBuffer, aVRAM );
346:        textPosX = textPosX + CHARCTER_SIZE;
347:        aStrings++;
348:    }
349: }
350:
351: void Off2VRAM( u32 aSourceAddress, u32 aDistAddress )
352: {
353: /*
354:     int x, y;
355:     u16* ScreenBuffer;
356:     u16* offScreenBuffer;
357:
358:     offScreenBuffer = ( u16* )aSourceAddress;
359:     ScreenBuffer = ( u16* )aDistAddress;
360:
361:     for( y = 0;y < SCREEN_SIZE_Y;y++ ){
362:         for( x = 0;x < SCREEN_SIZE_X;x++ ){
363:             ScreenBuffer[y * SCREEN_SIZE_X + x] = offScreenBuffer[y *
SCREEN_SIZE_X + x];
364:         }
365:     }
366:
367:
368: */
369:         // DMA3
370:     REG_DM3SAD = aSourceAddress;  // Source
```

```
371:        REG_DM3DAD = aDistAddress;              // Destination
372:        REG_DM3CNT = 0x84000000 + 38400 / 2; // 転送ワード( 32,16bit )数
373:    }
374:
375:
376:    // Random number generator
377:
378:    u32 gRandomSeed;
379:
380:    /*
381:    ---------------------------------------------------------------------------
382:     Global variables
383:    ---------------------------------------------------------------------------
384:    */
385:    static u32 cur_value;
386:
387:    /*
388:    ---------------------------------------------------------------------------
389:     Initialize ramdom number generator
390:    ---------------------------------------------------------------------------
391:    */
392:    void seed_lc( u32 v )
393:    {
394:        cur_value = v;
395:
396:        return;
397:    }
398:
399:    /*
400:    ---------------------------------------------------------------------------
401:     Linear congruential method random number generator
402:    ---------------------------------------------------------------------------
403:    */
404:    u32 random_lc( void )
405:    {
406:        /* 2^31 = 2147483648 */;
```

```
407:     cur_value = ( 1103515245UL * cur_value + 12345 ) %
     2147483648UL;
408:
409:     return cur_value;
410: }
411:
412: void InitUART( u16 parameter )
413: {
414:     REG_RCNT = 0; // SIO Enable
415:     REG_SIOCNT = 0; // SIO Reset
416:     REG_SIOCNT = UART_MODE
417:         | SEND_ENABLE
418:         | RECV_ENABLE
419:         | CTS_ENABLE
420:         | DATA_LENGTH_8
421:         | parameter;
422: }
423:
424: void SendByte( u8 data )
425: {
426:     REG_SIODATA8 = data;
427: }
428:
429:
430: void Wait( u32 aWaitingTime )
431: {
432:     while( aWaitingTime-- );
433: }
434:
435: void AWait( )
436: {
437:     while( 1 ) {
438:         WaitForVsync( );
439:         if( !( *KEYS & KEY_A ))
440:             break;
441:     }
```

```
442: }
443:
444: void GetTitle( s8 *aGameTitle )
445: {
446:     u16 i;
447:
448:     s8 *GameTitle = ( s8 * )0x080000A0;
449:
450:     for( i = 0; i < 12; i++ ){
451:         aGameTitle[i] = GameTitle[i];
452:     }
453:
454:     aGameTitle[i]='¥0';
455: }
456:
457:
458:
459: #endif
```

TeamKNOxLib 関数一覧（番号はソースのLine No.）

■グラフィック関数（グラフィックライブラリ）

void SetPixel(u16 aX, u16 aY, u16 aColor, u32 aVRAM);	091～097
void DrawLine(u16 aX0, u16 aY0, u16 aX1, u16 aY1, u16 aColor, u32 aVRAM);	100～122
void DrawCircle(u16 aX, u16 aY, u16 aRadius, u16 aColor, u32 aVRAM);	125～156
void DrawBox (u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16 aColor, u32 aVRAM);	159～170
void DrawBoxHalf(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u32 aVRAM);	180～199
void BitBltSRCCOPY(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u32 aVRAM);	201～213
void BitBltOR(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u32 aVRAM);	216～228
void BitBltAND(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u32 aVRAM);	231～243
void BitBltExOR(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u32 aVRAM);	246～258
void BitBltMasked(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u16* aMaskImage, u32 aVRAM);	260～270
void BitBltMaskedComp(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16* aImage, u32 aVRAM);	272～301

void DrawTextInit( )	303～305
void DrawText(u16 aX, u16 aY, s8* aStrings, u16 aStringColor, u16 aBGColor, u16 aRectMask, u32 aVRAM);	308～349
void Off2VRAM(u32 aSourceAddress, u32 aDistAddress);	351～373
void DrawBoxEmpty(u16 aX, u16 aY, u16 aWidth, u16 aHeight, u16 aColor, u32 aVRAM);	172～178
void WaitForVsync(void);	074～078
void GetRGB(u8 *arRed, u8 *arGreen, u8 *arBlue, u16 aPosScreen, u32 aVRAM);	080～088
■通信関数（通信ライブラリ）	
void InitUART(u16 parameter);	412～422
void SendByte(u8 data);	424～427
■文字列操作関数（文字列操作ライブラリ）	
void num02str(u8 number);	011～040
void num04str(u16 number);	042～071
■乱数関数（乱数ライブラリ）	
void seed_lc(u32 v);	392～397
u32 random_lc(void);	404～410
■ボタン操作関数（ボタン操作ライブラリ）	
void AWait( );	435～442
■時間関数（時間ライブラリ）	
void Wait(u32 aWaitingTime);	430～433
■その他の関数（その他のライブラリ）	
void GetTitle(s8 *aGameTitle);	444～455
■グローバル関数（グローバルライブラリ）	
s8 gWorkStr[8];	009
u32 gRandomSeed;	378

Game Boy Advance 実機動作
Actual Device Demonstration

中級者・上級者向け

これまではPC上のエミュレータで作成したGBAのプログラムを動作させていましたが、ここでは、PCからGBAに実際のソフトを転送して実機で動作させてみましょう。また、実機動作に必要なPCとの通信システム“ULA”の詳細解説を行ないます。

実機動作の必要性

GBAの最大の特徴はその携帯性です。小型·軽量なのでどこにでも持ち運ぶことができ、いつでも自由に遊ぶことができます。ここまでは、PC上のエミュレータでプログラムの動作確認を行なってきましたが、これには以下に述べる2つの問題があります。

① 本当に実機で動作するプログラムなのか？
② PC上では何のソフトを実行しているのかわからない

ということです。問題点①については、「う～ん、GBAのエミュというのだから、このエミュで動けば実機でも動くんじゃないの？」というくらいしか言いようがありません。これはエミュレータの再現性の問題とも絡んできます。

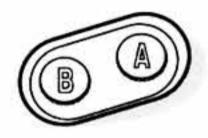
問題点②は、エミュレータの存在から説明しなければいけません。すなわち、エミュレータが存在しなければ動作しないプログラムをPC上で作成することに意味があるのかということです。

これらの問題点を解決するには話はすごく単純で、GBA本体に作成したプログラムを転送して動作させてみればいいのです。「百聞は一見にしかず」というわけです。

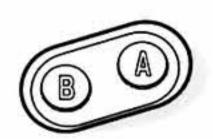
Chapter 06

GBA実機動作の歴史

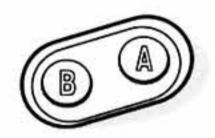
GBAで実機動作させるアプローチは世界各地で行なわれています。PC関連の技術やかつてのGBCなどで使われたコンセプトを用いたものが存在していたからです。これらの技術を応用したツールを利用して、作成したプログラムを実機に転送し、実機動作を行ないます。

このGBAへのプログラム転送ツールにも、第1世代〜第4世代までの変遷があります。

第1世代：パラレルポート経由FlashROM

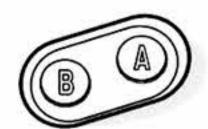
FlashROMを積んだカートリッジに対してパラレルポート経由で書き込みを行なうツールです。代表的なものにFlashAdvanceLinker（下写真）などがあります。

FlashAdvanceLinker

第2世代：マルチブート

マルチブートと呼ばれるメカニズムを用いた転送ツールです。代表的なものにJeff's MBV2（下写真）などがあります。

MBV II

第3世代：USB経由FlashROM

第1世代のパラレルポートの代わりにUSBを用いたものです。代表的なものにEZ-FA（下写真）などがあります。

EZ-F Advance

第４世代：USBマルチブート

第２世代のパラレルポートの代わりにUSBを用いたものです。代表的なものにULA(-FX)、F2A-UL(下写真)、USB-ブートケーブルなどがあります。

F2A-USB Linker

xLAプロジェクト(PLA, ULA)

国内で発足したGBAへのプログラム転送ツールとして認知されているものです。ULAとはUSB-Linker Advance、PLAとはParallel Linker Advanceで、xLAとはxにUあるいはPのどちらかが入るという意味です。xLAプロジェクトという名前でもわかるとおり、それぞれのシステムで共通のシステムプログラムやクライアント(コントロール)ソフトウェアを使えるようにしてユーザーの利便性を高めたものです。

ULAの動作原理の原型となっているのは、開発サイトで有名なJeffのMBV2やオプティマイズ氏のGBAブートケーブル、あるいはGBA SceneのPICを用いたケーブルです。その他には生DOSを用いてパラレルポートのタイミングを完全にコントロールしているシステムなどがあげられます。いずれにしてもパラレルポートです。

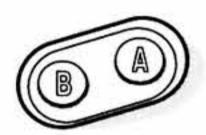
とくにオプティマイズ氏は、国内でいち早くGBAブートケーブルを使ったプログラムの実機転送を実現させていました。その成果物をとおしてオプティマイズ氏に通信ケーブルを介して実際のデータをGBAの内蔵RAMへダウンロードする手法を教えてもらったので、この方法を用いてULAを実装することになりました。

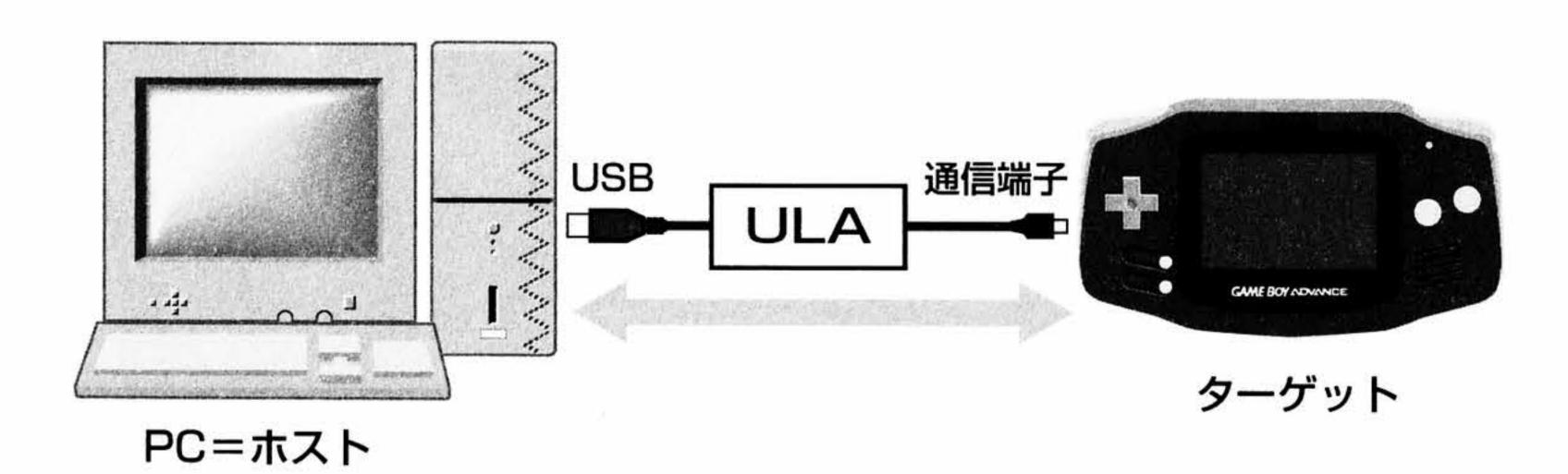
筆者は基本的にパラレルポートなどのレガシーデバイスに今後は注力しない方向で開発を行ないたいと思っていますので、USBを使うことにしました。

実機動作の原理

6-4-1 動作シーケンス

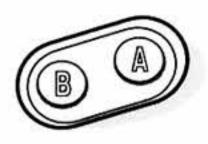
どのようにプログラムが実機で動作しているかを見てみましょう。

ここではULAの接続から、先に作成したGBAのプログラムがGBAにダウンロードされるまでを見ていくことにします。この原理が理解できれば、自分でULA相当品を制作してGBAソフトを実機で動作させることも可能になります。

6-4-1-1 全体図

全体の構成は下図のようになります。

ULA システム接続図

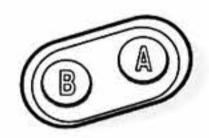
ULAを使用してPCとGBAを接続する

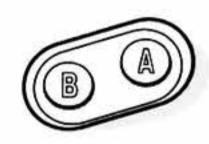
6-4-1-2 ファームウェアのダウンロード

ULAはそれ自体が8ビットのコンピュータシステムです。コンピュータなので、動作するにはプログラムが必要になります。電源投入時やシステムの起動に必要となるソフトウェアをとくにファームウェアと呼んでいます。このファームウェアにより、PC－GBAの通信が実現されます。

6-4-1-3 1.5 BIOS のダウンロード

この1.5 BIOSというのは、筆者が所属するTeamKNOxのULA開発チームが名付けたものです。この部分は元々、先に説明したPLA（ブートケーブル）の成果物で、PLA（ULA）を介して動作する最小限の通信プログラ

ムのことを指します。また、1.5 BIOSという名称の1.5ですが、これは、このBIOSの特徴を表わしています。

マルチブートで最初に起動するプログラムを1番目のBIOSということで1st BIOSと呼ぶと、その次に起動するプログラムは2番目ですから2nd BIOSになります。ではなぜ1st BIOSではなく1.5 BIOSと小数点を含んでいるかというと、じつは、このBIOSは単体で存在するのではなくファームウェアにバンドルされるカタチで使われるからです。そんな理由から1.5 BIOSと命名されています。

6-4-1-4 2nd BIOSのダウンロード

GBA本体に内蔵されているBIOSはブートを行なう大切なプログラムです。よって、これに敬意を表して、1st BIOS（最初のBIOSという意味）と呼ぶことにします。これに加えてカートリッジの吸出しやフラッシュの書き込みなど、ULAの機能の大部分を司るプログラムを2nd BIOSと呼んでいます。この2nd BIOSは別ファイルで提供されているので、これを差し替えることによって機能の追加が簡単にできるようになっています。

6-4-1-5 通信開始

ここまでのやり取りでPC⟺GBAの通信ができるようになりました。あとは自作のソフトをダウンロードするだけで、実機で遊ぶことができるようになります。

Chapter 06

ULAの製作

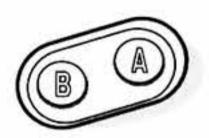
ULAは商用製品ではありません。手に入れるには基本的に自作する必要があります。ここではその自作方法について説明します。

自作の方法は基本的に2つあります。1つは、USBチップの評価ボードを購入してそれにケーブルを付け足して実現する方法です。元々、評価基板は何か回路なりケーブルをつけることが前提になっているので、非常に加工しやすくできています。ですから、ここではこの方法を初心者用として位置づけています。ただ、工作しやすい代わりに、少し高価なことと回路自体が大きめになってしまいます。これは元々、評価ボードの特性ですからどうにもなりませんが、工作のしやすさは初心者にとってはうれしいことだと思います。

もう1つの実現方法は市販品を改造することです。これは本来、違う用途で開発されたものをULAとして利用するものです。これらはジャンクとして出回っていることが多く、びっくりするほど安く入手することが可能です。

初心者向け(AN2131SC[EZ-USB]評価ボードを利用)

ULAの機能はUSBチップのAN2131SCで実現されています。このチップは資料や開発ツールの入手しやすさから、アマチュアからプロまで多くのユーザーに支持されています。また、USBの学習用としても優れているので、多くのキットや完成品が評価基板として売り出されています。

これらを用いるとULAだけではなくUSBそのものの学習にも役立ちます。ここでは市販されているものを中心に紹介します。

6-5-1-1 IPI社製評価ボード

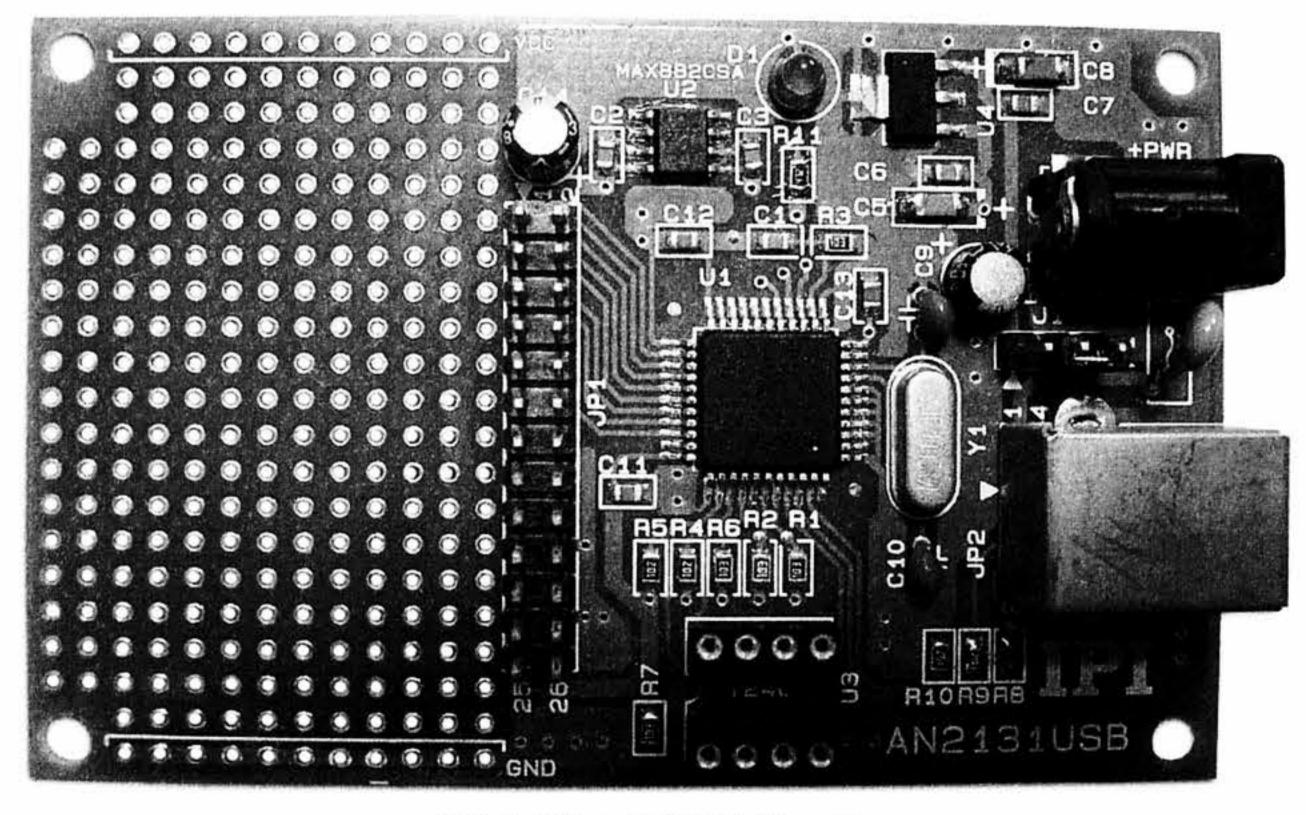
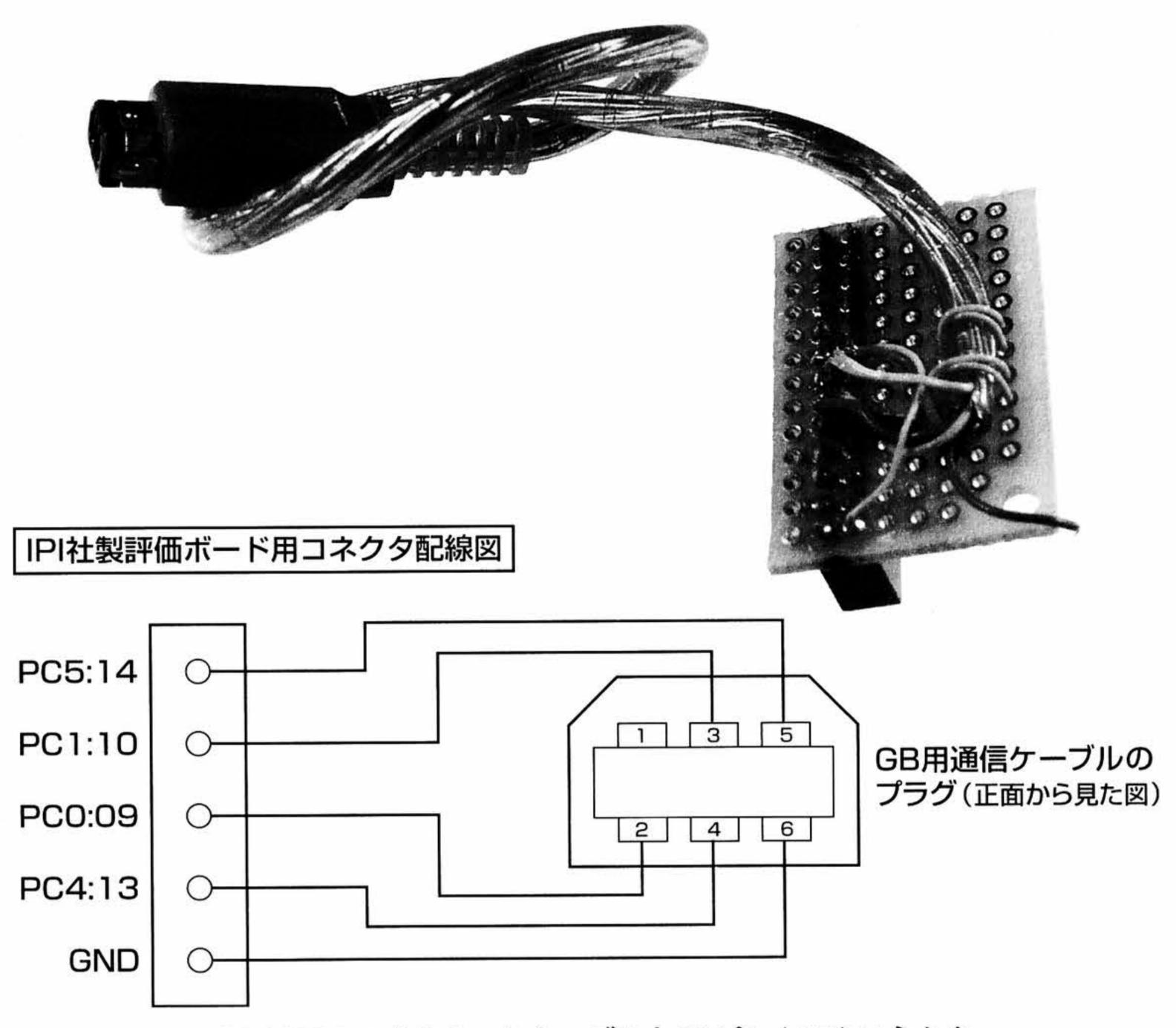
完成品です。前述のとおり、USBの評価・実験のためのボードです。プロのエンジニアがプロトタイピングするためのものです。つまり、完成品で動作保障(ULAとしての動作保障ではないので念のため)をしているので自分の目的(ターゲット)をすばやく実験することができます。

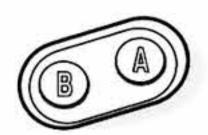
IPI社製USB評価ボード

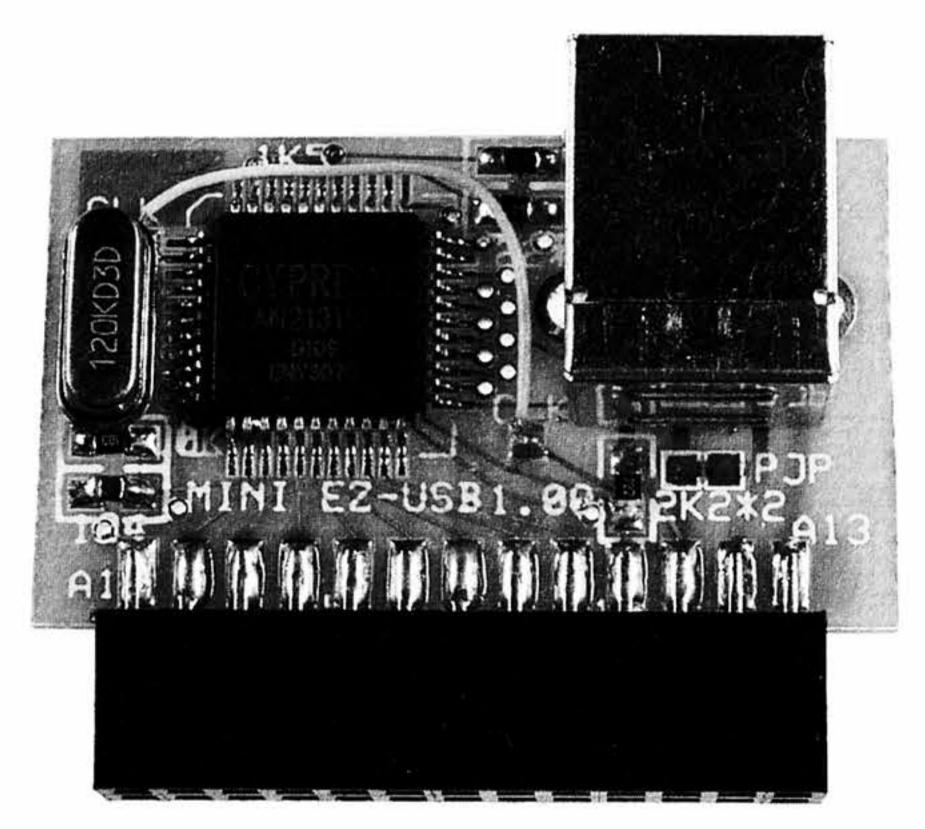
IPI社製評価ボード用コネクタ配線図

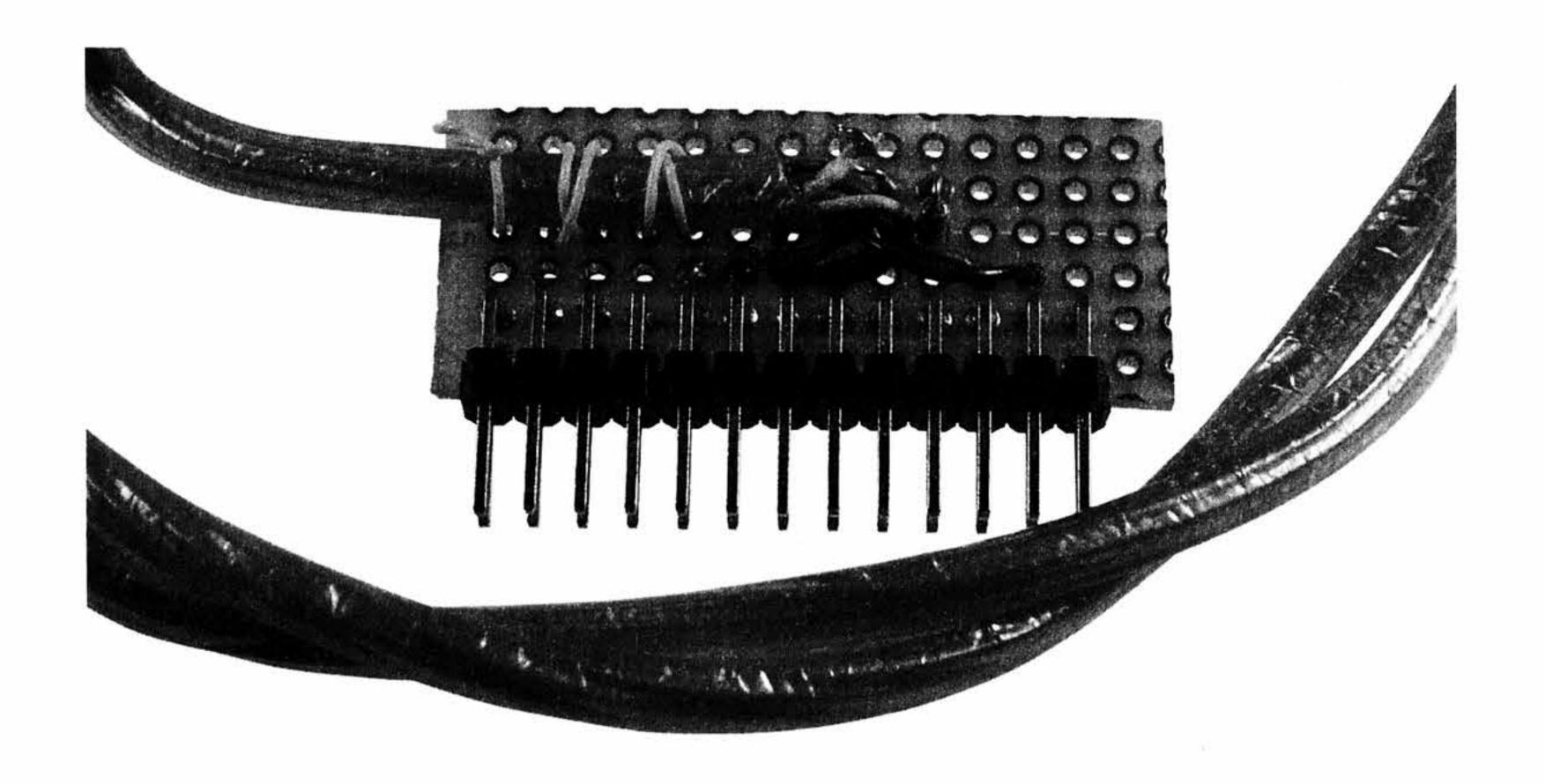
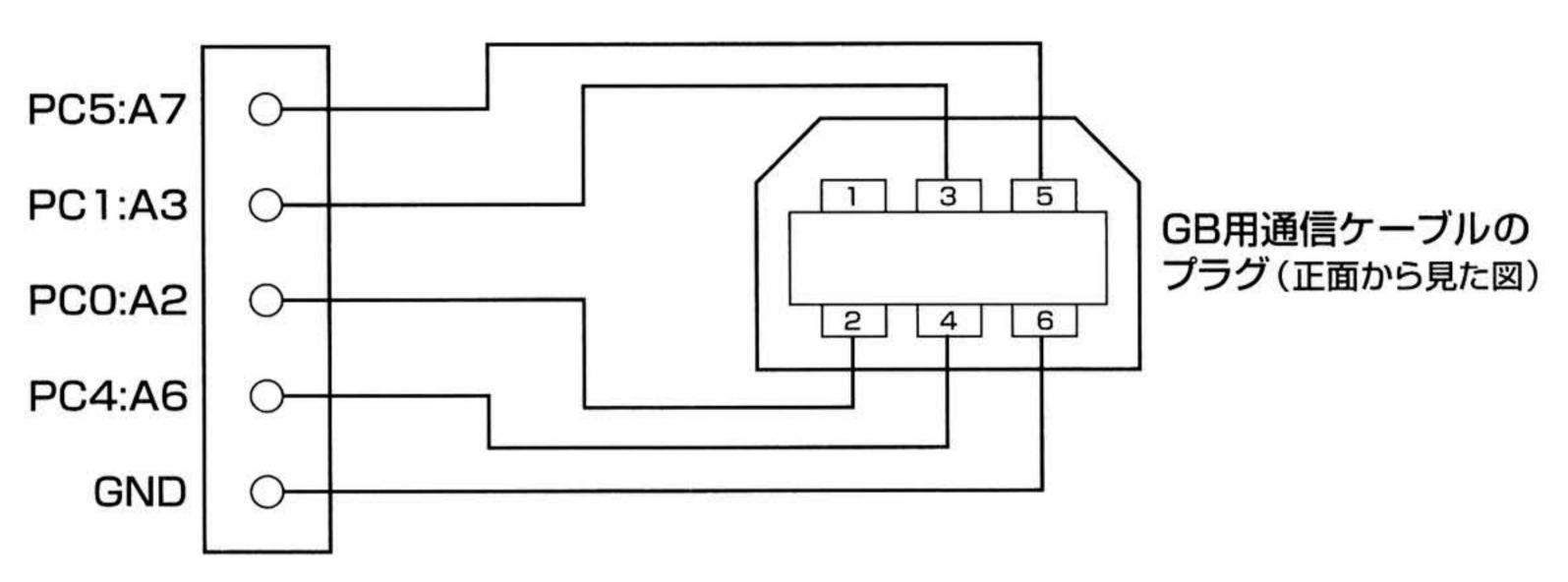
PC5:14
PC1:10
PC0:09
PC4:13
GND

1 3 5
2 4 6

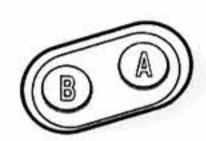
GB用通信ケーブルのプラグ（正面から見た図）

GBC/GBAコネクションケーブルと26ピン（メス）コネクタ

GBC/GBAアダプタボード＋IPI社製USB評価ボード＋GBA

6-5-1-2 Mini-EZ-USB（EZ－ULA）

先述したオプティマイズ氏が企画したEZ-USBのキットです。IPI社製のものに比べると安価ですが、さまざまな部品が省略されています。やや中級者向けと言えるでしょう。0.8mmピッチの表面実装チップや抵抗をハンダづけしなければいけないので、ハンダづけのテクニックが多少必要になります。

Mini-EZ-USB

Mini-EZ-USB用ケーブル&コネクタ

Mini－EZ－USB用コネクタ配線図

GBC/GBAコネクションケーブルと26ピン（オス）コネクタ

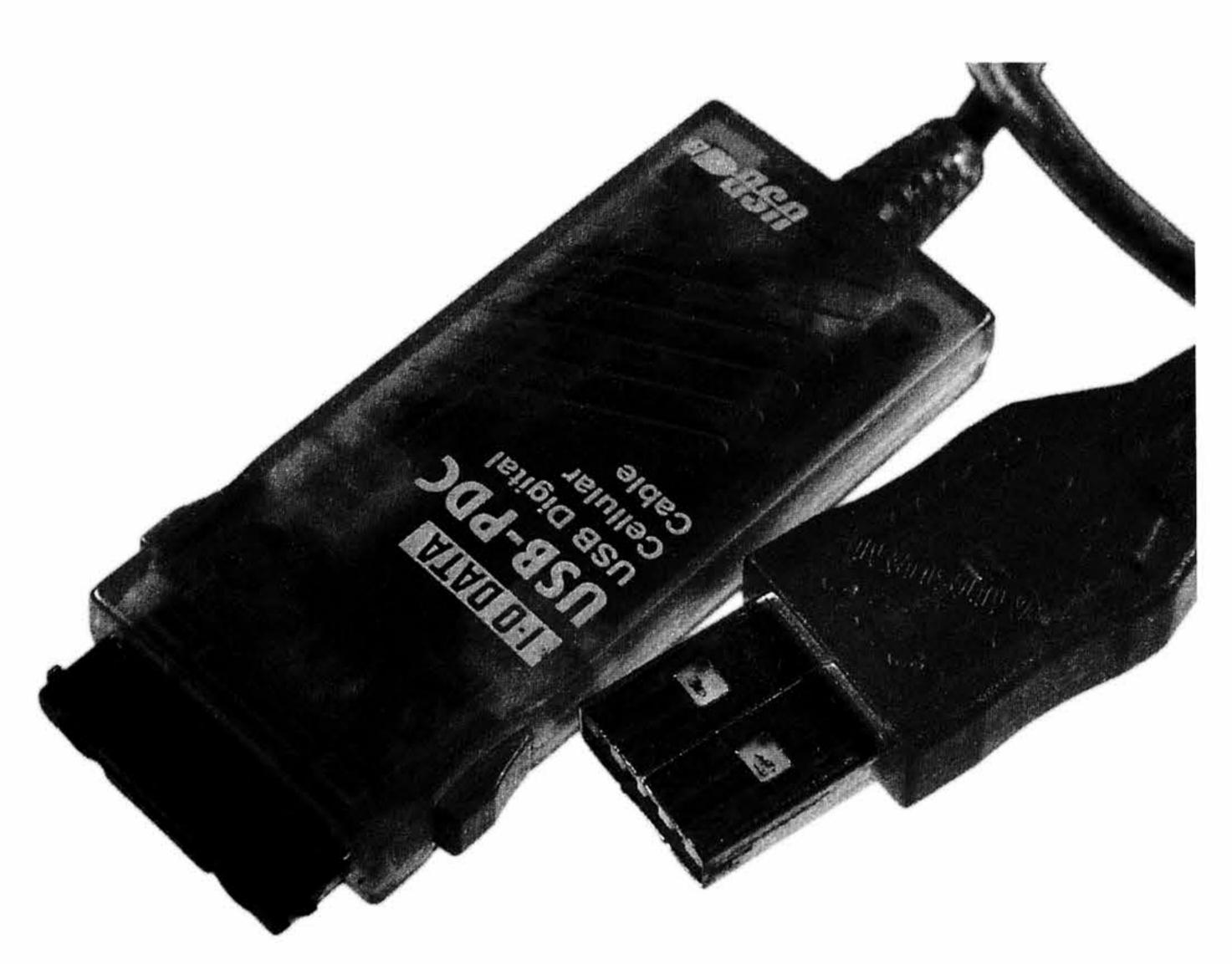
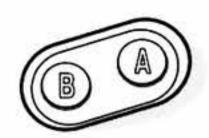
上級者向け（市販製品の改造）

EZ-USBを用いた製品は数多くリリースされています。これらの製品は一般に高価ですが、たまにジャンクやアウトレットなどで放出される場合があります。そのときは反対にびっくりするほど安価です(50円～!!)。ここでは、いままでに私が手がけた改造を中心に紹介します。

6-5-2-1 USB-PDC

USB-PDCは元々、PCと携帯電話(PDC)を接続するためのインターフェイスケーブルです。このなかの中核部品としてEZ-USBが用いられています。まだ、USBの評価基板がそれほど売り出されていなかった頃は、このUSB-PDCでUSBの勉強をしました。なるべく、USB-PDCの機能を保持したままで使うことに腐心しました。

ただし、最新版のULAはUSB-PDCは動作対象外になっているので注意が必要です。ただ、これも最新版の回路図に合わせて本体の改造を行なえば問題なく動作します。大変コンパクトなので腕に覚えのある人はこちらのほうがスッキリしたものができると思います。

これの後継機としてUSB-PDC2なるものがありますが、こちらは使用チップが違うのでULAにはなりません。くれぐれも注意してください。

USB-PDCパッケージ（I・O DATA製）

USB-PDCの本体部（この中の基盤を流用します）

6-5-2-2 CE-PD03

CE-PD03は、ULAが現在の地位を確保するに至った最大の功労者といえるでしょう。秋葉原や日本橋を中心に出店している中古に強い某パソコンショップでシャープ製ノートパソコン「メビウス」の周辺機器であるCE-PD03が100円で大量放出されました（最安値は50円という情報もありますが、筆者は未確認です）。

専用周辺機器は通常、その対象となるPCを持っていないと意味がないものですが、ここでは周辺機器として使うのではなく、その基板が搭載しているチップが目当てです。このCE-PD03はULAの中核部品であるAN-2121SCが使われています。インターネットなどの報告を見ると一部ではAN-2131SCが使われているようです。

サイプレスによると2121と2131の大きな違いは、ファームウェア格納用の内蔵RAMの容量の違いです。ちなみに2121では4KB、2131は8KBになっています。この容量の違いは致命的です。というのもULAのファームウェアであるgba_boot.bixは既に4KBを軽く超えています。今後も機能の拡張に伴い、容量は増加していくかもしれません。

Mebius コーポレートモデル周辺機器一覧 - Microsoft Internet Explorer

http://www.sharp.co.jp/products/share/html/option.html

PC-PJ140K/PC-PJ140L 専用周辺機器		
増設用SDRAMボード[144pin SO-DIMM]	64MB CE-M544B	希望小売価格 35,000円(税別)
	128MB CE-M545B	希望小売価格 70,000円(税別)
デジタル携帯電話アダプター	CE-PD02	希望小売価格 15,000円(税別)
cdmaOneアダプター	CE-PD03	希望小売価格 15,000円(税別)
最大24倍速CD-ROMドライブユニット	CE-CD03	希望小売価格 20,000円(税別)
USB接続FDドライブユニット	CE-FD04	希望小売価格 12,000円(税別)
大容量バッテリー	CE-BL10	希望小売価格 28,000円(税別)
シートバッテリー(リチウムポリマ)	CE-BP02	希望小売価格 21,000円(税別)
ACアダプター	CE-J03V	希望小売価格 10,000円(税別)

- 本体付属の標準バッテリーと大容量バッテリーを同時に装着することはできません。また、大容量バッテリーとシートバッテリーを同時に装着することもできません。
- 本体付属のデジタル携帯電話アダプターとcdmaOneアダプターを同時に接続することはできません。
- デジタル携帯電話アダプターに使用できる携帯電話の機種については、にて順次ご案内しています。

◀トップページへ戻る

PC-PJ150L 専用周辺機器		
増設用SDRAMボード[144pin SO-DIMM]	64MB CE-M544B	希望小売価格 35,000円(税別)
	64MB CE-M554B	オープン価格
	128MB CE-M545B	希望小売価格 70,000円(税別)
	128MB CE-M555B	オープン価格
デジタル携帯電話アダプター	CE-PD02	希望小売価格 15,000円(税別)
cdmaOneアダプター	CE-PD03	希望小売価格 15,000円(税別)
最大24倍速CD-ROMドライブユニット	CE-CD03	希望小売価格 20,000円(税別)
USB接続FDドライブユニット	CE-FD04	希望小売価格 12,000円(税別)
大容量バッテリー	CE-BL10	希望小売価格 28,000円(税別)
シートバッテリー(リチウムポリマ)	CE-BP02	希望小売価格 21,000円(税別)
ACアダプター	CE-J03V	希望小売価格 10,000円(税別)

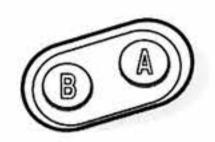
- 本体付属の標準バッテリーと大容量バッテリーを同時に装着することはできません。また、大容量バッテリーとシートバッテリーを同時に装着することもできません。
- 本体付属のデジタル携帯電話アダプターとcdmaOneアダプターを同時に接続することはできません。
- デジタル携帯電話アダプターに使用できる携帯電話の機種については、にて順次ご案内しています。

◀トップページへ戻る

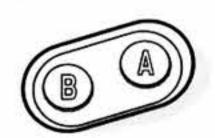
インターネット

シャープのホームページでは、CE-PD03が当時希望小売価格15000円で販売されていたことがわかります

改造方法

ここではCE-PD03を例にした改造方法を説明します。今後もCE-PD03が放出される可能性があるとは断定できませんが、改造方法のノウハウを会得しておくのは無駄ではないと考えます。また、この改造方法は前記の評価基板にも簡単に応用できると思います。

6-5-3-1 2121で動作した!?

前述のとおり、CE-PD03にはULAの中核部品であるAN-2121SCが使われています。しかし、ULAのファームウェアgba_boot.bixは既に4KBを軽く超えていますし、今後も機能の拡張に伴って容量は増加していくかもしれません。8KBの内蔵RAMを持つAN-2131SCが使われているCE-PD03なら問題ないのですが、ネットの住人たちは大量に放出されたおいしいモノを目の前にただ、指をくわえて見ているだけではありませんでした。もしかして、2121でもgba_boot.bixが使えるのではないかと人柱の名乗りをあげるユーザーが出てきたのです。

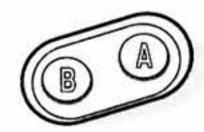
最初は誰もが無理を承知で、2121で製作したULAにファームを流し込んだのです。しかし、なぜか使えました。使えるという報告が相次ぎました。そこで内蔵RAMの容量測定用ソフトをリリースして、2121でも使えるかどうかを調べるようにしたのです。

6-5-3-2 で、ワンコイン…

USB-PDCのジャンクや携帯電話のケーブルなど、いままでも単発での安い出物はありましたが、これだけ大量の放出はありませんでした。あ

る程度の再現性もあるようです。そこで、500円玉ひとつでできるワンコインULAとして発表されるに至りました。内訳として…

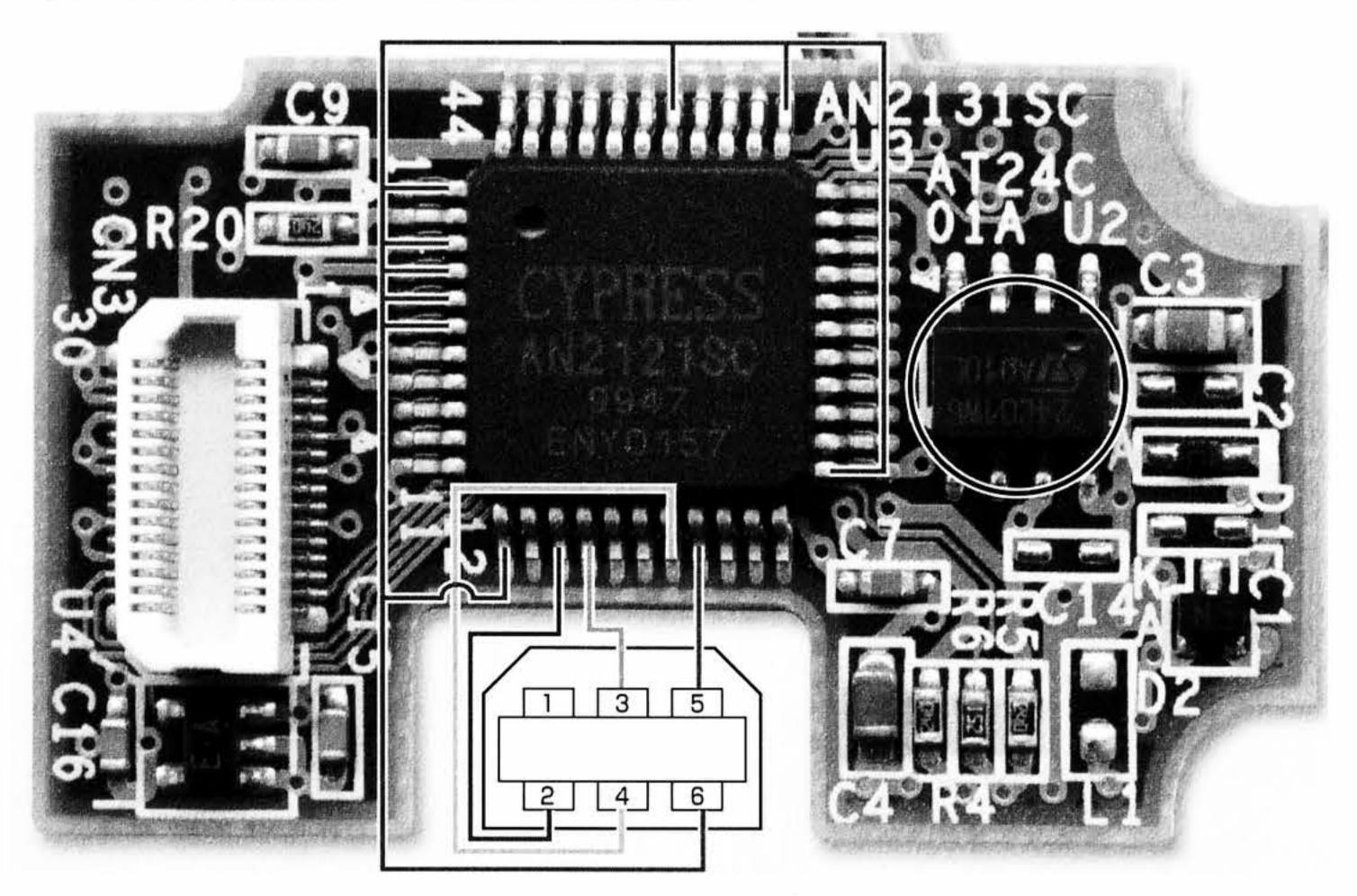
パーツ価格の内訳	
CE-PD03	100円
USBケーブル	100～200円
GBケーブル	100～200円
ケース	0円（CE-PD03か廃物利用）

こんな感じです。USBケーブルとGBケーブルは手持ちがあればタダですが、新規に購入する場合は、価格の変動が大きいのでワンコインで納まらなくなる可能性もあります。注意が必要です。

その他の項目としては、製作用の各種の道具類（ハンダこてやニッパなど）があげられます。新規に購入するとそれなりの出費になってしまう場合もあります。そういう場合は、学校・職場の友人・同僚の工作の得意な人に頼むという手もあります。部品を2セット買っても1000円程度で済みますから、作ってくれる人の分も購入して、一緒に作ってもらうようにしてはどうでしょうか？

6-5-3-3 実際に作ってみると…

筆者自身も持ち運びできる小さいULAが欲しかったので、1つ作ってみることにしました。筆者自身は幸運なことにUSBケーブル、GBケーブルなどは既に手持ちがあったので、CE-PD03のみを購入するだけで済みました。ケースは手元にあったお菓子のケースを使うことにしました。これでさえ同僚にもらったのでタダです。つまり、材料費は100円ということです!!

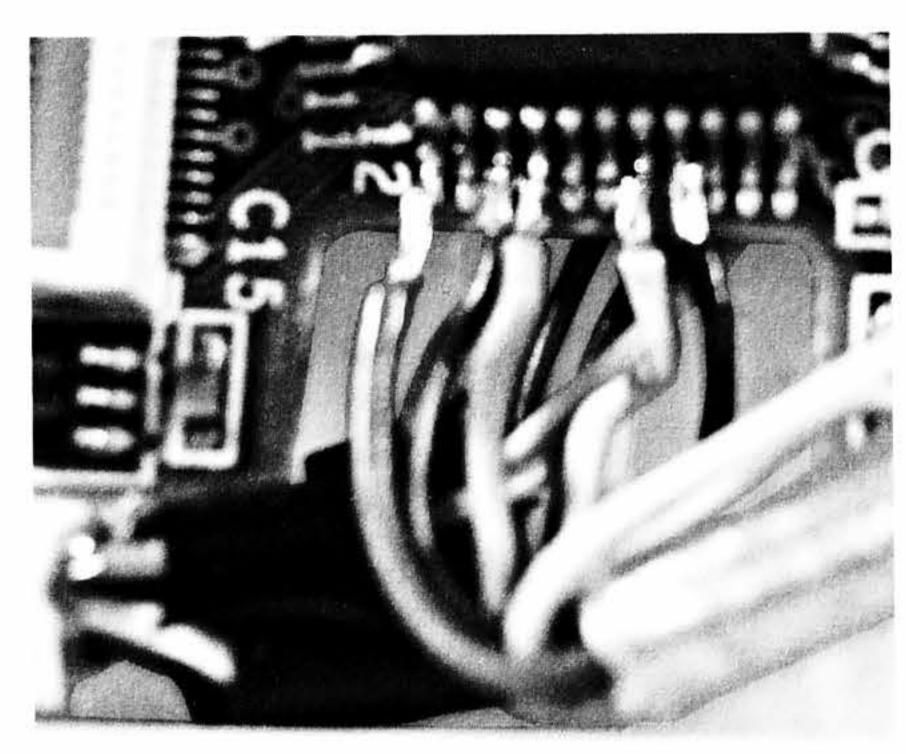
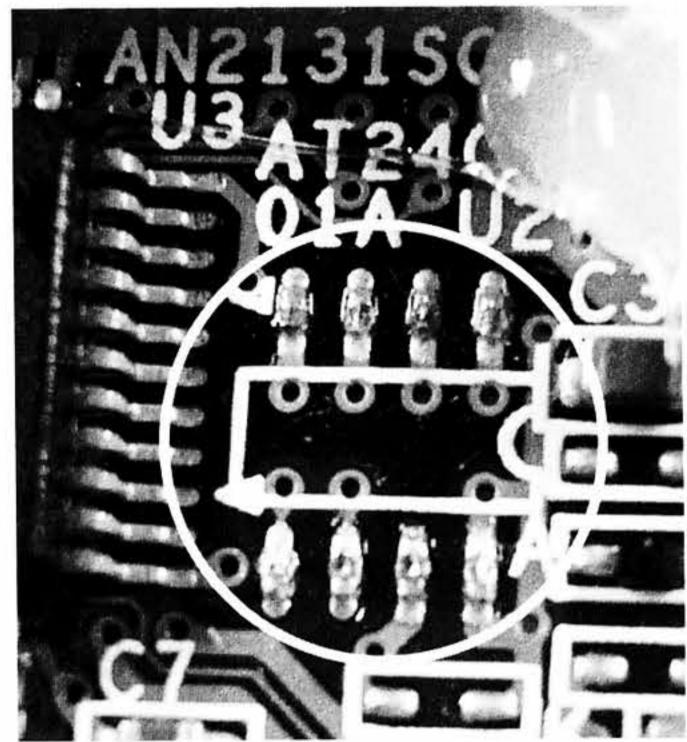
手順1：分解

まず最初にCE-PD03を分解して基板を取り出します。48ピンのICが載っている基板が今回利用する基板です。

AN-2121SCにGBC/GBAコネクションケーブルを図のように取り付けます

手順2：ケーブルの加工

USBケーブルとGBケーブルを適当な長さに切断します。筆者は両方とも20～30cm程度で使っています。3mm程度をワイヤストリッパで被覆を剥いてハンダメッキします。

手順3：基板の加工

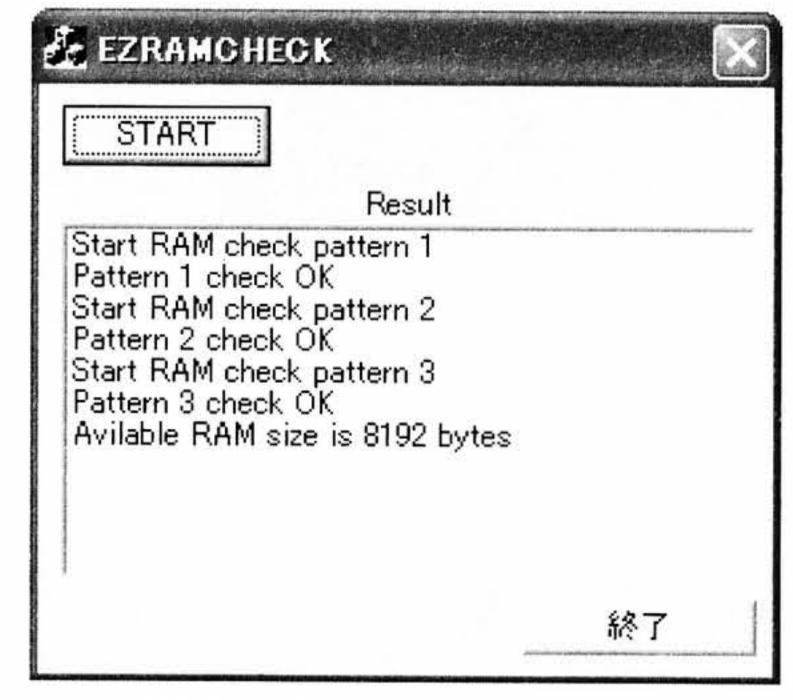
AN21x1ではEEPROMをファームウェア用として搭載しています。ezusbw2k.infファイルを書き換える方法とEEPROMを取り外す方法があります。面倒なので筆者はデバイスの足を全部ニッパで切断しました。このEEPROMを活用したい人はezusbw2k.infファイルを書き換えて使うといいでしょう。

ケーブルの取り付け（左）とEEPROMの除去（右）の様子

手順4：USBケーブルの配線

基板にはミニUSBコネクタが接続されています。このミニUSBコネクタを切断して、その代わりに手順2で加工したUSBケーブルを配線します。USBケーブルの配線の色は決まっているようです。以下のとおりになります。配線チェックは充分に行なってください。

CE-PD03 USBケーブルの色分け		
VCC	茶	赤
DATA	赤	白
DATA+	橙	緑
GND	黄	黒

手順5：RAM容量の確認

先に紹介したRAM容量確認ソフトでRAMの容量を確認します。図2のようになればOKです。この作業が終了すれば、この基板がULAとして動作できそうかどうかがわかります。

EZRAMCHECK（図1：START）

EZRAMCHECK（図2：Result）

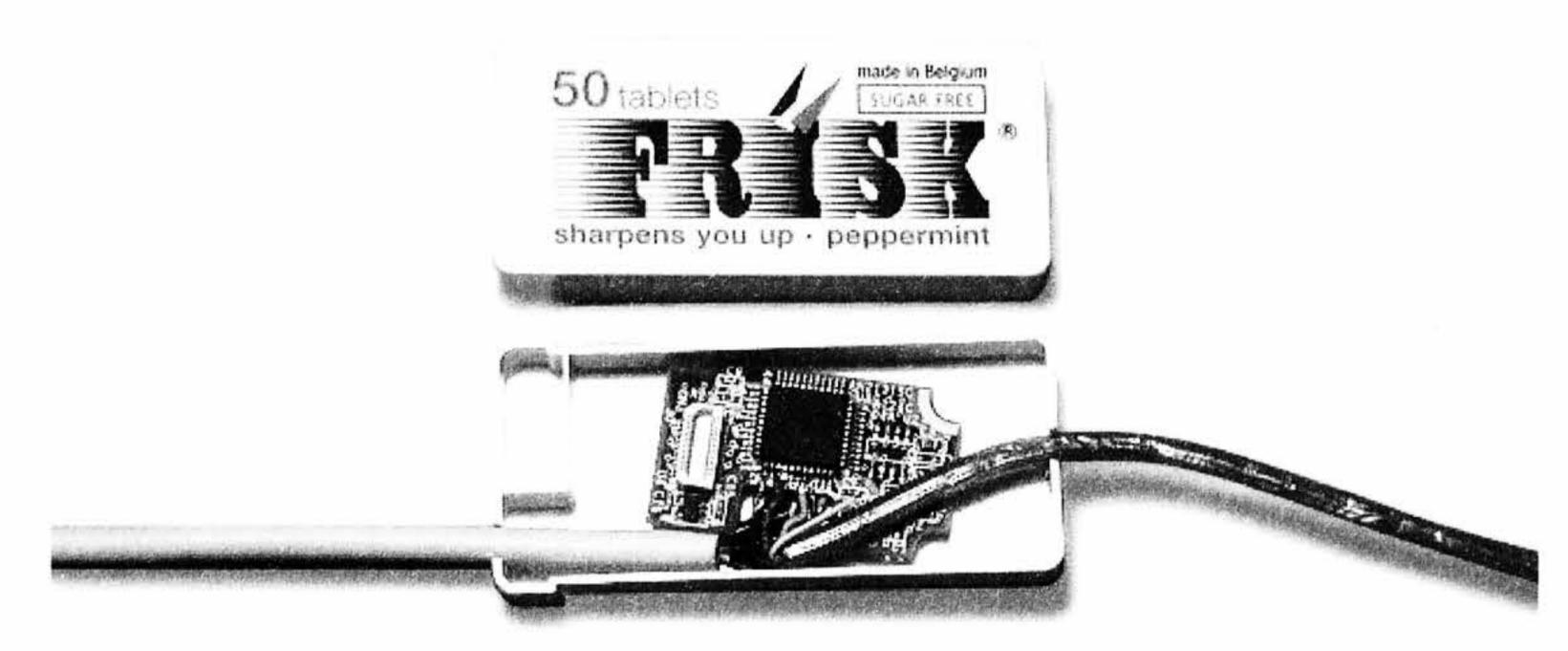
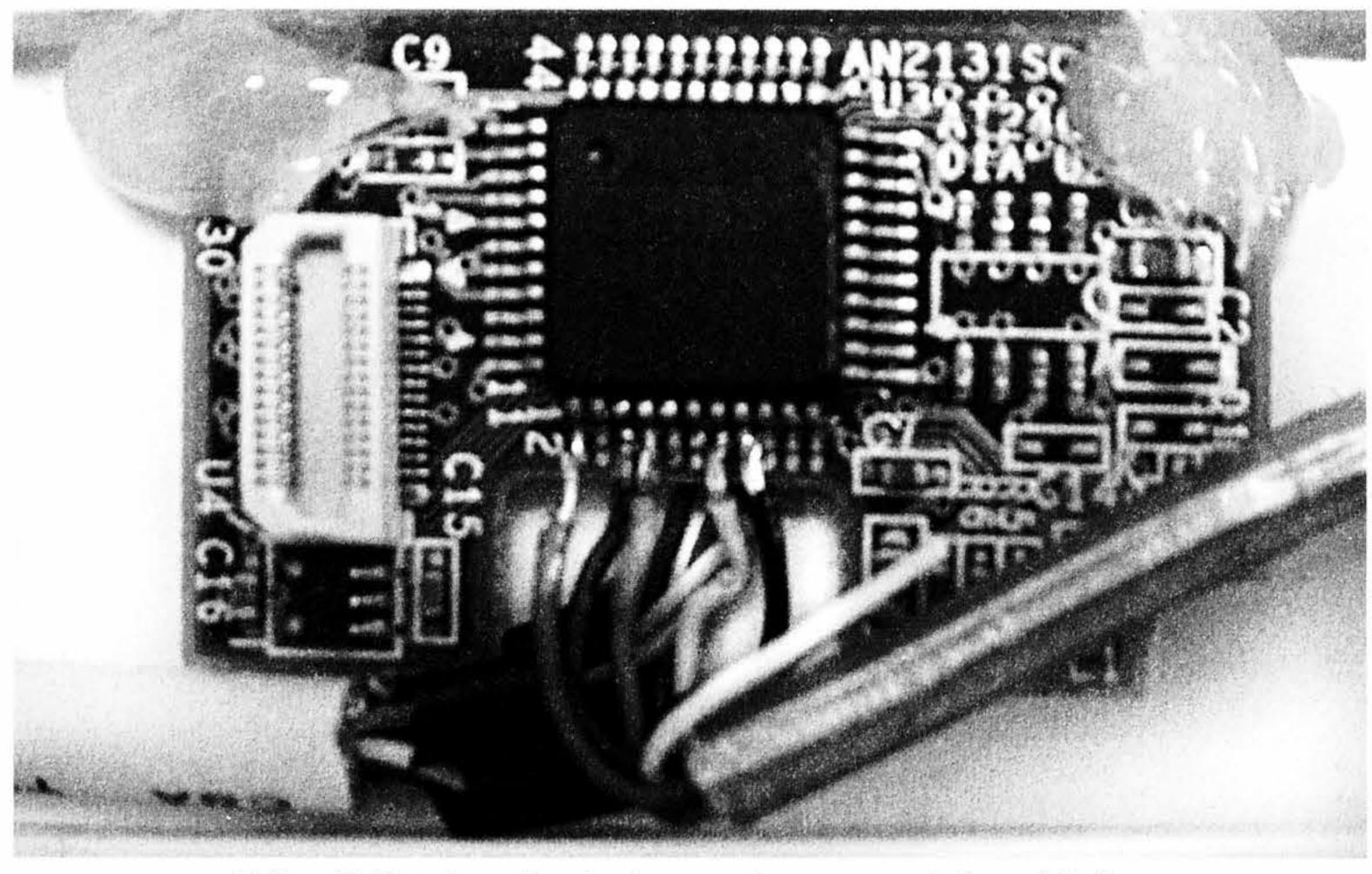
手順6：GBケーブルの配線

ここからが本番です。かなりハンダづけの腕が要求されますが、苦労する甲斐はあります。170ページの図のとおりにハンダづけしてください。

色が指定していないのは、GBケーブルはモノによって配色が違うためケーブルの色とピン番号が対応しないためです。配線前にテスタで確認して配色を判断しておきます。今回はULAのみとして利用するためPB0の配線はありません（単なる手抜き…？）。

GBケーブルの配線をテスタでチェック（ハンダづけする前にもチェックします）

手順７：動作確認

配線チェックは充分に行なってください。さらにもう１回の配線チェックがあなたのPC、ULA、GBAを破壊から防ぎます。手元にカートリッジがある場合はFlashManager、ない場合はULA-GPやBONSAI-WAREなどでチェックしてみてください。

USB-PDC（**CE-PD03**）**を使って作成したULA**（**お菓子のケースに入れました**）

最後に基盤やケーブルをボンドでケースに固定すれば完成！

自作プログラムの転送

手順７まで終わった人、あるいはULAの完成品を手に入れた人は自作プログラムを実機へ転送できます。とくに自作のULAの場合、この動作確認は最も緊張することと思います。

プログラムの転送にはTeamKNOxで作成したULA-HostV2を使います。ULA-HostV2はその名のとおりULA-Hostのバージョン２になります。ULA-HostはULA（ハード）の発表と共にリリースされたソフトです。このソフトは筆者がはじめてVC++で組んだソフトで、開発が終了してからかなり月日がたっています。筆者自身はULA-Hostのリリース以来、それなりにVC++での開発に取り組んだこともあり、歩みは遅いですが何とかプログラミングスキルは向上してきました。その視点でULA-Hostを眺めると結構イタイ点も出てきました。また、ULA関連のソフトはオプティマイズ氏作のfwlibを利用させてもらっていますが、こちらのバージョンもあがってきて、その意味でも現状とは合わなくなってきたことから、バージョンアップに踏み切ったものです。

ULA-HostV2

最初に作ったULA-Hostは、とにかく動作することが重要でした。プログラムの構造や効率よりも動作を優先したわけです。今回のV2では安定動作に加えて、次の5つのポイントを追求するようにしました。

①効率

似たような処理を共通化しました。とくにV1ではデータやプログラムで別々に実装されていたダウンロードやアップロードの処理などを、まず一本化しました。これにより、ソースリストそのもの、ひいては実行コードの小型化が実現できました。

②安定性

ULA本体への転送はスレッドを用いました。スレッドを用いることによりUIとULAへの転送部分が並列して動作するようになりました。これにより任意のタイミングで転送をキャンセルできるようになり、操作性が向上しました。

③操作性

内容的には②とも若干かぶっています。各動作の起点はビットマップのボタンを利用して、直感的に操作できるようにしてあります。また、V2は主に開発者のために開発されました。そのため、コマンドラインからも起動できるようになっています。動作はEWRAMへの転送だけですが、開発者（筆者？）はこの機能を主として利用するので、この機能を実装することにしたのです。

⑤ 拡張性

今回、最も考慮したのがこの部分です。V2のプログラム構成は大きく2つに分かれています。起動時の画面や操作するUI部分（UI）とULAへの転送部分（エンジン）になります。このエンジンをクラスライブラリにしたことで、さまざまなプログラムに適用可能（組み込む）になっています。同時に公開したThingy-ULAでは、この転送部分を組み込んであります。

⑤ 互換性

V2でできることと操作性などは以前のバージョンとほぼ同様にしました。これにより以前からのユーザーもそのまま操作することが可能です。

また、fwlib.bin、gba_boot.bixなどのシステムファイルは、後述のFlashManagerと一部、合わせてあります。

基本的な操作

それぞれの操作はボタンを押して決定します。以下の機能があります。

① プログラムの転送 (Download program to GBA/FlashCart)

この機能は、自作や吸い出したプログラムを内部RAMやFlashCartへ転送する機能です。とくにULA-Hostの最大の機能が内部RAMへのプログラムの転送です。これにより自作のプログラムなどの容量の小さいもの（例：BONSAI-WAREなど）は、とくにFlashCartがなくても手軽に楽しむことができます。

V1では内部RAMとFlashCartの選択は自動になっていましたが、現在ではfwlibの動作によりマニュアルで選択するようになっています。基本的な操作は右ページのようになります。

(1) V2の起動
(2) コマンド選択
ULAHost...
ファイルを開く
ファイルの場所(I): Reversi20030218
bitmaps
TeamKNOx
ReversiAdv.mb.gba
ファイル名(N): ReversiAdv.mb.gba
ファイルの種類(T): GBA binary file (*.gba)
読み取り専用ファイルとして開く(R)
開く(O)
キャンセル
ヘルプ(H)
(3) ファイル選択
(4) GBAの電源OFFを促す
Project xLA
Turn off GBA !!
OK
(6) メモリ選択-内部のEWRAMを使うかフラッシュカートリッジを使うかどうか?
ULAHostV2
Download to EWRAM of GBA ?
はい(Y)
いいえ(N)
(5) GBAの電源を切ったかどうかの確認(OKクリック)
(8) プログラム転送の転送開始
GBA Turn On Waiting ...
Turn On GBA with [SELECT] + [START]
ULA System Information
Data Transfer
Data Transfer ... 98 % Completed
Abort
Cart Information
Memory Type : EWRAM
Memory Size : 0Mbit
BackUp Memory Type : NONE
Close
(7) [START] + [SELECT] を押しながら電源再投入
(9) 転送後実行(自動)

②セーブデータのカートリッジへの転送(Download data to Cart)

PCに吸い上げたセーブデータ(カートリッジ内のバックアップメモリ内容)を再びカートリッジに書き戻すコマンドです。V2特有の機能として、*.gba、*.mbなどのプログラムも選択可能になっています。これは後述するブートスティックを使うためです。

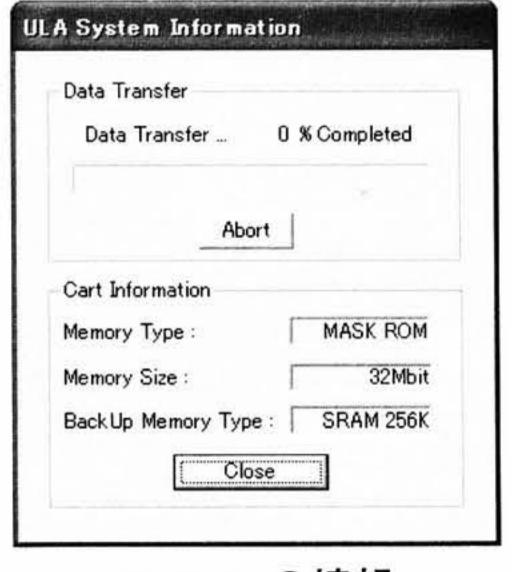
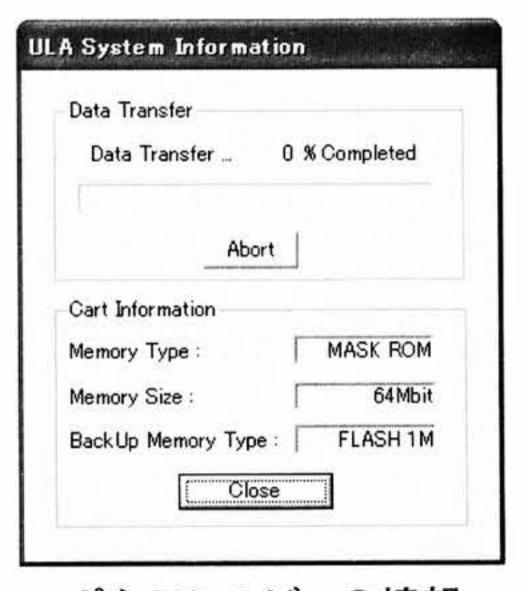
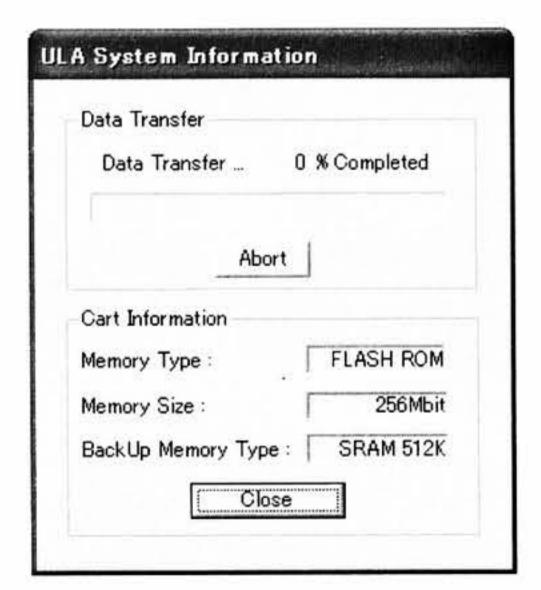
③プログラムのPCへの転送(Upload program from cart)

カートリッジ内のプログラムをPCへ転送します。吸い上げたプログラムをGBAエミュレータ(VBAなど)を使ってPCで遊ぶことができます。

④セーブデータのPCへの転送(Upload data from cart)

カートリッジのセーブデータをPCへ転送します。

⑤カートリッジの解析(Get Cart Information)

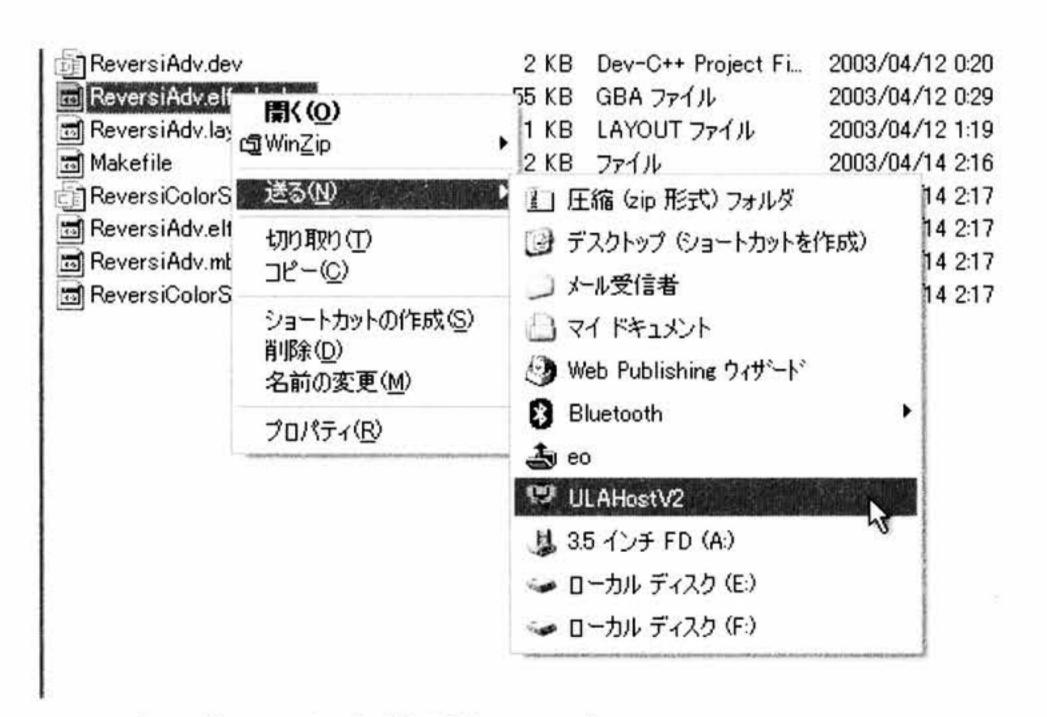
どのようなカートリッジが使われているかを解析して表示します。

F-Zeroの情報

ポケモンルビーの情報

フラッシュカートリッジの情報

たとえばF-Zeroの場合はマスクROM(量産型の書き直しできないROM)で容量は32Mbit(8MByte)、バックアップメモリはSRAM 256Kbit(32KB)であることがわかります。

Chapter 06

開発者向けの機能

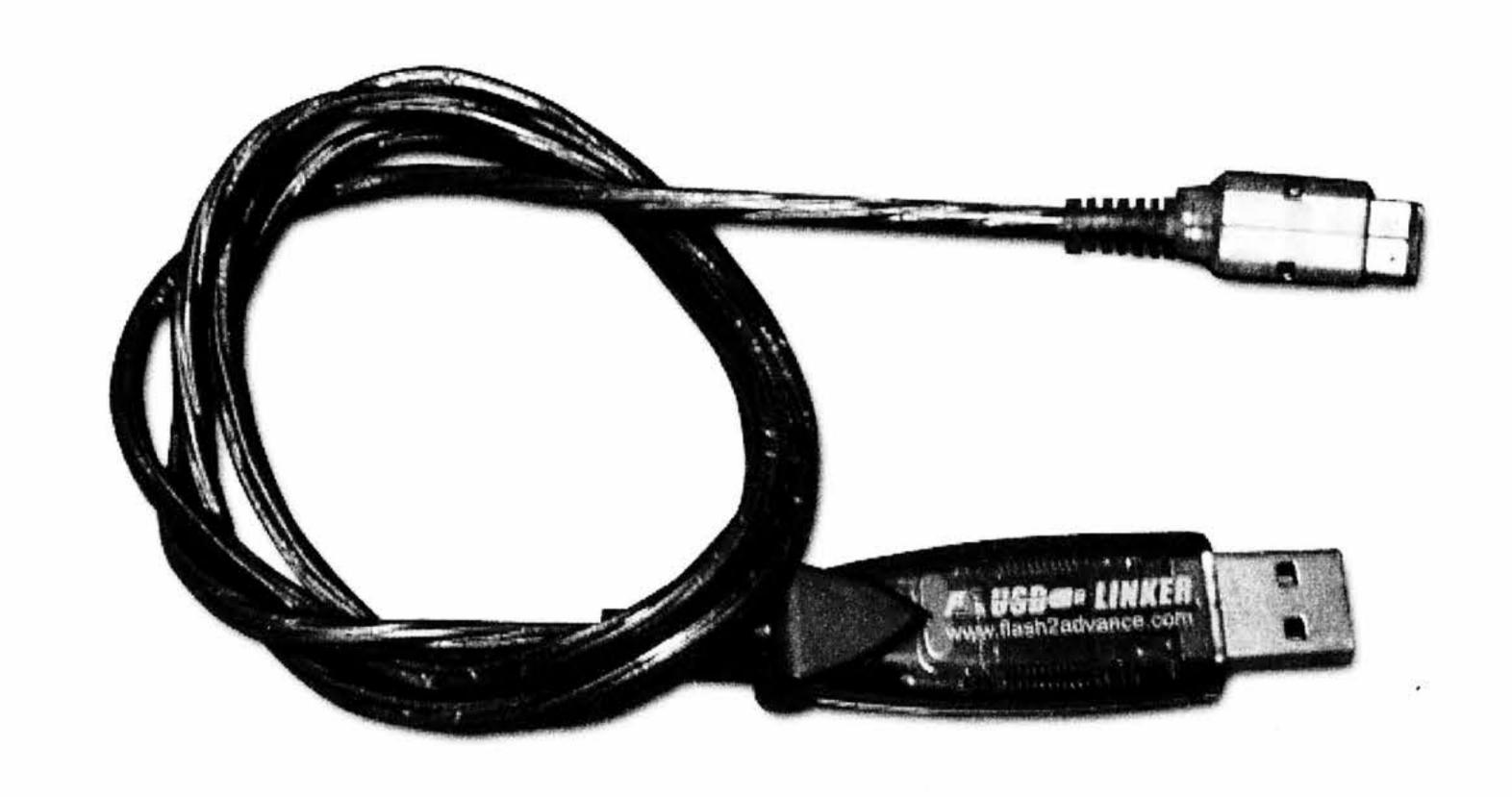
次に開発者向けの機能を見てみましょう。これはV2に搭載されている機能です。GBAの開発はDOS窓を使ったものが主流です。とくにmakefile中にさまざまなコマンドを記述するわけです。このような状況ではコマンドラインで起動できるプログラムが便利になります。また、GBAのソフト開発はそれほど大きなサイズのものでなければ、プログラムはEWRAM 256KB領域へ展開すればこと足ります。そこでEWRAMへの展開に限り、コマンドラインでもプログラムを起動できるようにしました。また、この機能を実装すると対象ファイルを選んで右クリックの「送る」メニューが使えるようになります。生成されたオブジェクトを簡単にULAで転送できるようになります。

プログラムを直接選択して右クリックメニューからULA経由でEWRAMに転送する

その他のツールたち (ULA以外の第4世代のツール)

F2A-USB Linker

ULAが発表されてしばらくしてから出てきたバックアップツールです。構成チップの類似性からULAのパクリではないかと言われましたが、構成が若干違います。ULAとF2A-USBの違いは先駆者の調査によって基本的にはPC5とPC6の違いだけであると確認されています(ロットによっては違うかもしれません…)。そこで、現状ではPC5を出力しているULAのファームであるgba_boot.bixを改造してPC6でも出力できるようにFlashManagerの作者でもあるMootan氏が改良を加えました。

一部のユーザーはこのことに早くから気づき、PC5とPC6をブリッジ(ショート)して以前のファームでも対応できるようにしていました。ところが、PC5とPC6の両出力だと、ブリッジしている場合は出力同士がぶつかることになって回路的にはよくありません。つまり、F2A-USBを無改造で使っているユーザーは問題ありませんが、ブリッジさせているユーザーは以前のファームを使うか、改造前の状態に戻す必要があります。

いずれにしてもULA関連で開発された資産(FlashManager、ULA-HostV2、ULA-GP、Thingy-ULA)を、量産品であるF2A-USBで使うことができるようになりました。ハード製作は苦手だけど、ULA関連ツールを使ってみたかったというユーザーには朗報だと思います。

ブートケーブルUSB

実機転送ツールのパイオニアであるブートケーブルのUSB版です。転送速度では居並ぶ競合を抑えてダントツの1位です。後述するULA-GP相当の機能も取り込んでいるので、GBAをお手軽にゲームパッドにすることもできます。価格は若干高めですが、ハードの製作には興味がないという人には最適でしょう。

ULA-FX

ULAの高速版です。FlashManagerの作者であるMootan氏が製作されました。基本的にはULAと同様の構成ですが、処理プロセッサに48MHzの高速タイプのデバイス(FXプロセッサ)を用いています。ファームウェアのチューニングにより、ULAの2倍〜4倍程度の処理速度を実現しています。今後、注目のツールです。

ULA-FXの試作基盤

Column

ハンダづけのコツ

GBケーブルにハンダメッキが十分してあれば、とくにハンダを後づけしなくても大丈夫です。後づけする場合もほんの少量で大丈夫。できれば極細のハンダを盛れば、隣とのブリッジを防げます。

Column

ULA の入手方法

ハンダづけがどうしてもできなかったり、時間がないという人がULAを入手する方法として、ネットーオークションで入手したり、回路製作を代行してくれるヘルパーさん的な人に頼むという方法があるようです。そうした方法で入手するのも1つの方法であろうとは思います。ただ、自分で作ったULAの愛着はひとしおであることは間違いありません。

Column

電子工作について

筆者は男女差別主義者ではありませんが、少なくとも男性ならハンダづけくらいは軽くできたほうがいいような気がします。ハンダづけが得意な男がセクシーかどうかは別にしてもです。

同様に料理や裁縫なども軽くこなせるヒトであるほうがいいと思います。最近は隣国から安価で良質な工業製品が大量に輸入されているため、自分で作る行為自体が減少し、何もかもあまりにも安易に入手できてしまうことは問題です。自分の手を動かすことがないので、モノの本当の価値がわからないヒトが増えているように思います。自分の手で作り上げることで、モノ作りというものに、いかに多くの工数がかかるかを実感したほうが、何かを開発する上でも、プログラミングを作る上でも、そしてヒトとして生きていく上でも絶対にいいのです。

Column

エミュレータの再現性

昨今ではさまざまな機種のエミュレータが公開され、その機能と再現性を競い合っています。エミュレータとは「emulate」が語源ですが、emulateを新英和中辞典で調べると「…と競う、張り合う、熱心に見習う、まねる」などと記載されていました。なるほど、元々競い合うのが宿命のようです。

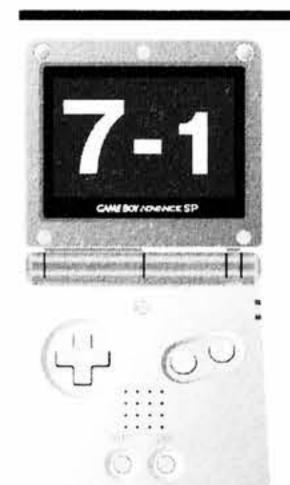
エミュレータは高度なプログラミングの知識と実装能力を必要とする領域にあるプログラムです。各ターゲット（たとえばWindowsでGBAをエミュレーションする場合は、WindowsとGBA）に精通していないと作れません。各ターゲットへの理解の深さが、基本的にエミュレータの再現性となって現われてくるわけです。

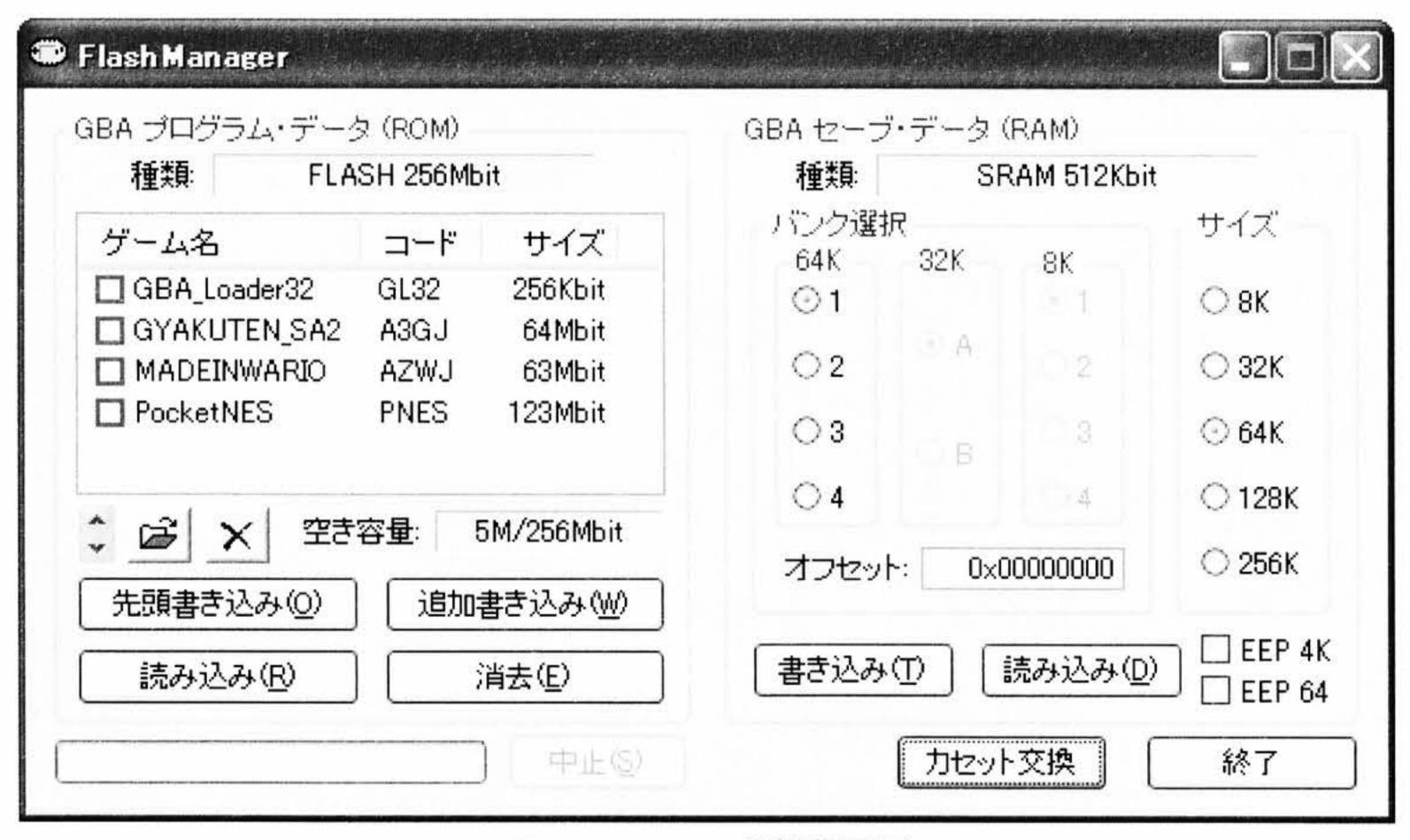
エミュレータの熟成度を見ると、ギクシャクしていたソフトが滑らかに動作するようになったり、最初は発声しなかったのが、版を重ねるにつれて重厚な発声が行なわれたりします。これは各ターゲットのグラフィックや発声メカニズムをより深く理解して実装したということに他なりません。筆者もぜひ、この領域にいつかチャレンジしたいと考えています。

Game Boy Advance（ゲーム ボーイ アドバンス）活用事例

Case of Exploitation

ユーザー主導

ULAの最大の特徴は草の根レベルで開発され、コミュニティがしっかりしているところです。さまざまな使いこなしテクニックやノウハウが共有され、自分流の使いこなしや制作事例がWebなどで発表されています。その中のいくつかを紹介しましょう。

FlashManager

これを使いこなしの範疇に入れていいものかどうか悩みます。ただ、ULAの普及や発展とともに成長して現在に至っているので、ここで紹介しておきます。FlashManagerは、市販の書き込み機とほとんど遜色ないレベルに仕上がっているソフトウェアです。また、ULA-FXもサポートしています。数本のゲームを1本のフラッシュカートリッジにまとめて収録したいときには大変便利です。

FlashManagerの操作画面

GBAのBIOSの吸い上げ

筆者が現在、活用しているGBA用のエミュレータはVBAです。VBAを利用していて、実機と決定的に違うことを実感するのが起動画面です。実機であれば電源投入時にGameBoyのロゴがレインボーカラーで鮮やかに現われますが、エミュだとそそくさとカートリッジのプログラムがはじまります。なんとかエミュでも、このオープニング画面を再現したいものです。ここではこのオープニング画面を実現するために、BIOSを吸い上げる方法とエミュの設定方法について説明します。

BIOSって？

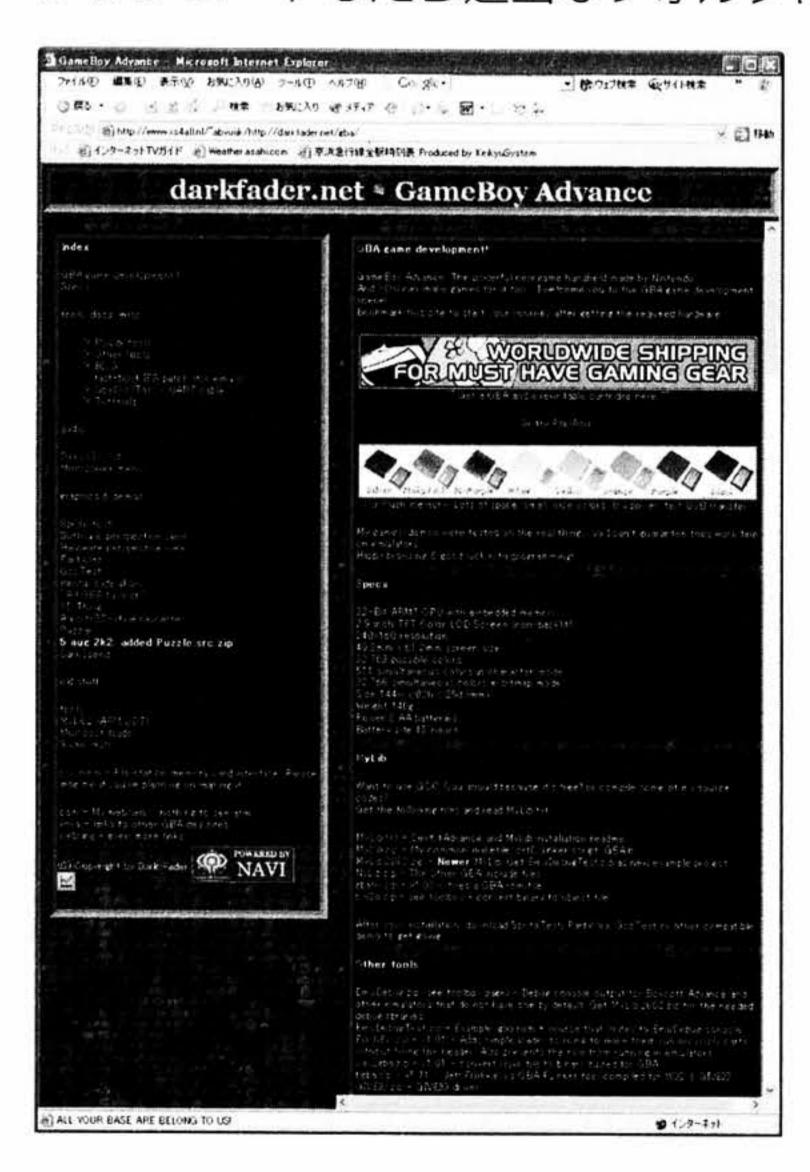
BIOSとはBasic I/O Systemの略称です。筆者がはじめてこの言葉に出会ったのは、たしか富士通の8ビットパソコンFM-8だと思います。BIOSはハードウェアをコントロールする最低限のシステムプログラムを指します。同じ文脈でファームウェアという言葉が使われることもありますが、慣習としてBIOSはその上に更に大規模のシステムプログラム（OSやミドルウェア）が載ることを前提として使われ、ファームウェアはそれ単体でシステムを動作させる場合に使われるような気がします。GBAにもこのBIOSが搭載されています。電源投入時のオープニング画面などのコントロールをはじめマルチブートなどを実現するために使われています。

BIOSはハードウェアをコントロールするためのプログラムですから、非常に基本的で有用です。なかにはBIOSを積極的に利用しているソフトもあります。

BIOSの吸い上げプログラム

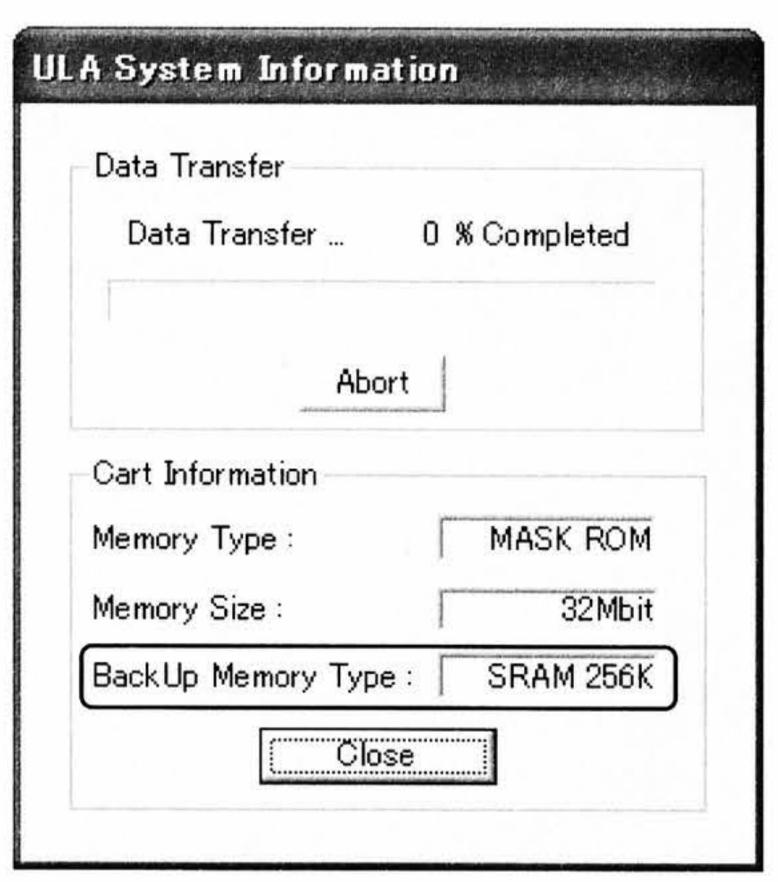
BIOSは極めて重要なプログラムですので、簡単にアクセスできないようにプロテクトされています。ところがこれを読み出すプログラムが既に作成され公開されています。

http://www.xs4all.nl/~abvuijk/http://darkfader.net/gba/

上記のサイトにアクセスして、dumprom.zip をダウンロードします。ダウンロードしたら適当なフォルダに解凍しておきましょう。

左側のIndexからBIOSをクリックします。すると右側にBIOS欄が表示されるので、DUMPROM.ZIPを探してクリックするとダウンロードできます。

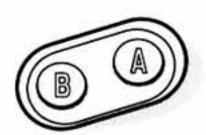
BIOS吸い上げの原理

dumpromはBIOS領域をカートリッジのバックアップエリアにコピーします。BIOSのサイズは基本的に16KBなので、これ以上のバックアップ領域を持つカートリッジが必要になります。筆者はF-Zeroのカートリッジを使ってみました。これは256Kbit（32KB）の領域を持っています。

今回使用したF-ZEROのカートリッジ

BackUp Memory Typeを調べておく

準備するもの
1. dumprom.mb.gba
2. ULA
3. ULA-HostV2
4. 適当なゲームカートリッジ
5. バイナリエディタ

作業をはじめる前に、左のリストにあるハードウェア/ソフトウェアを用意しておく必要があります。

1～4までは説明の必要はないと思います。問題となるのは、多くの人があまり利用する機会のない5でしょうか。

7-2-3-1 バイナリエディタ

エディタは通常、プログラムや文書の作成（入力）に利用されていると思います。バイナリエディタはその名のとおり、バイナリコードを入力・編集するためのツールです。Vectorやその他のダウンロードサイトで「バイナリエディタ」というキーワードで検索してみれば、たくさんのバイナリエディタを発見できます。いくつかダウンロードしてみて、自分の好みに合ったものを選べばいいでしょう。筆者はBZ-Editorを愛用しています。以前にWZ-Editorを利用していたこともあり、UIが似ていてとっつきやすかったからです。

ダウンロードサイト「Vector」で好みのバイナリエディタを探そう!

BIOSの吸い上げ実践編

それでは、実際にBIOSの吸い上げに挑戦してみましょう。

●手順1：ハードの接続

① ULAをPCに接続

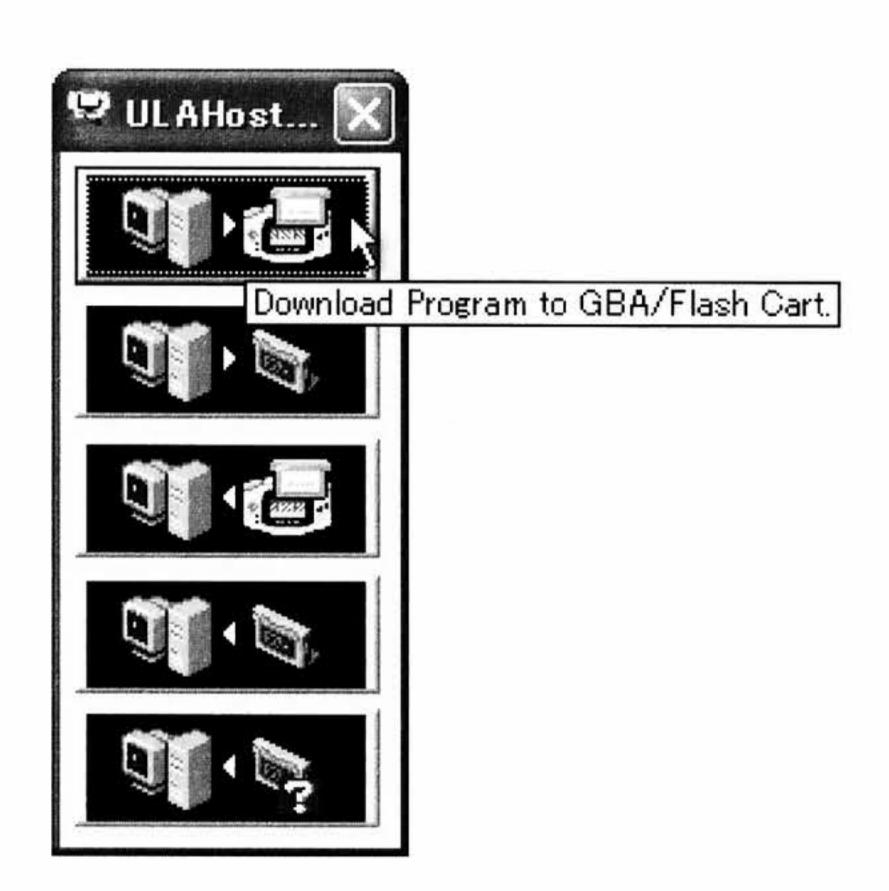
② カートリッジをGBAに装着

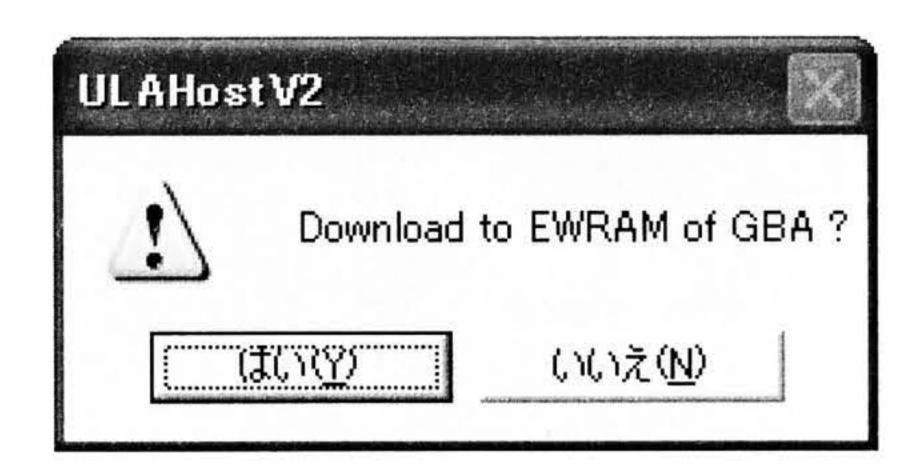
③ ULAとGBAを接続

●手順2：ソフトの立ち上げ

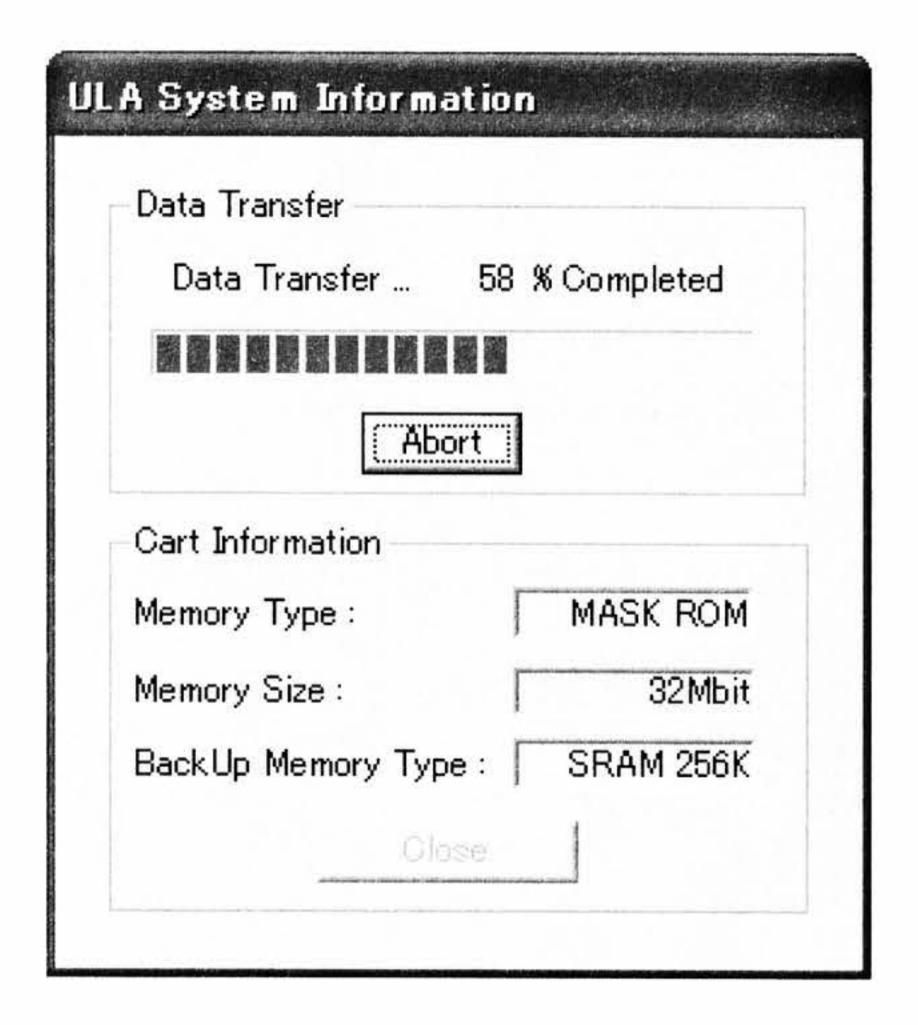
○ ULA-HostV2を起動

●手順3：プログラムの実行

① ULA-HostV2の最上段にあるメニュー(Download to GBA/Flash Cart)を選択

② DumpRom.mb.gbaを選択します。

③今回のプログラムはマルチブートモードで動作するのでDownload to EWRAM of GBAで「はい」を選択します。

④画面の指示に従ってGBAの電源をOFFにします。

⑤[START]と[SELECT]を押しながら電源をONにします。

⑥通常のマルチブートのソフトのようにダウンロードされて、自動実行します。

●手順4：プログラムの動作

○プログラムが動作すると、まず、画面全体が赤に変わります。その後、バックアップが終了すると緑に変わります。緑色に代わったのを確認したらGBAの電源をOFFにします。

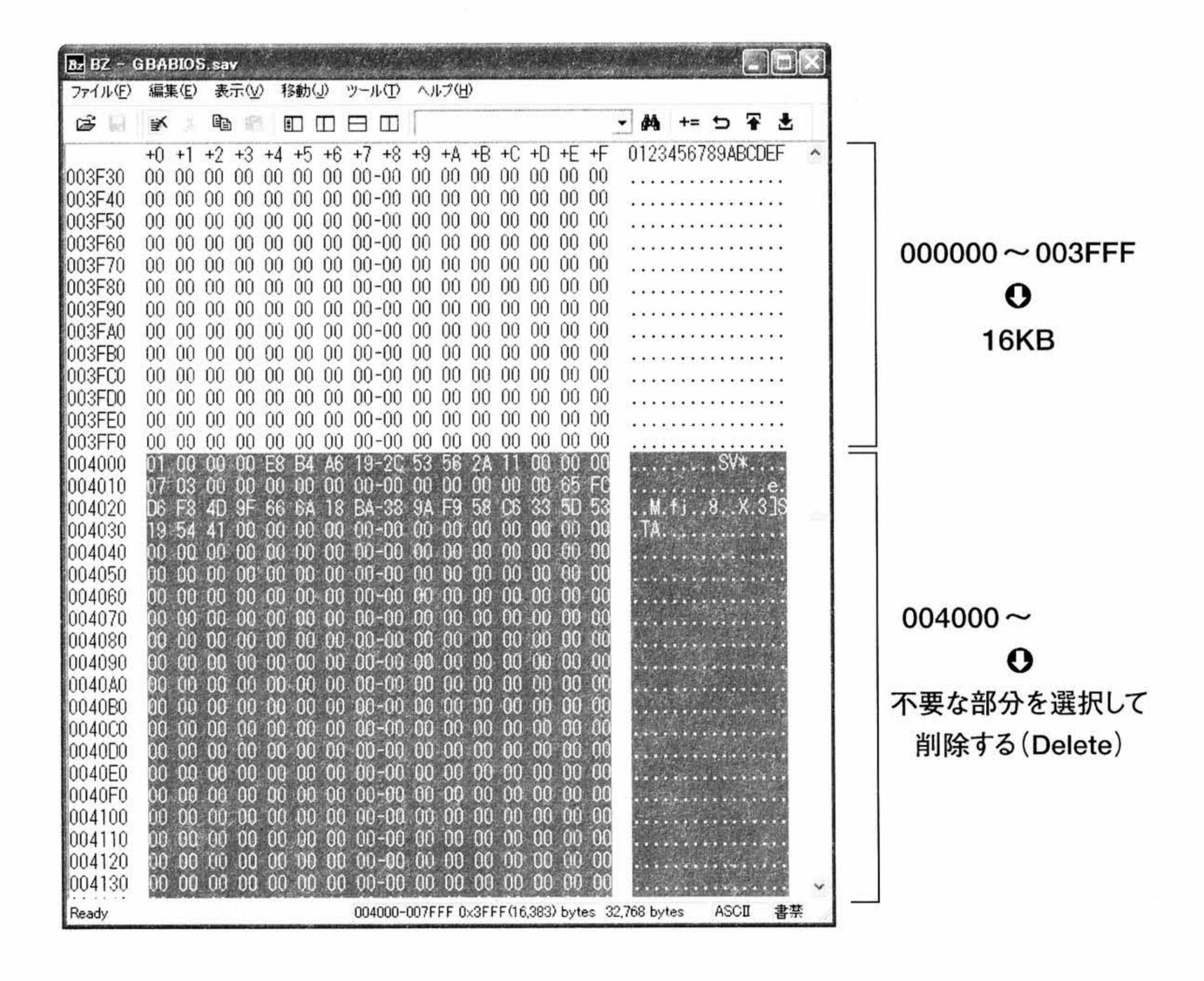
●手順5：データの吸い上げ

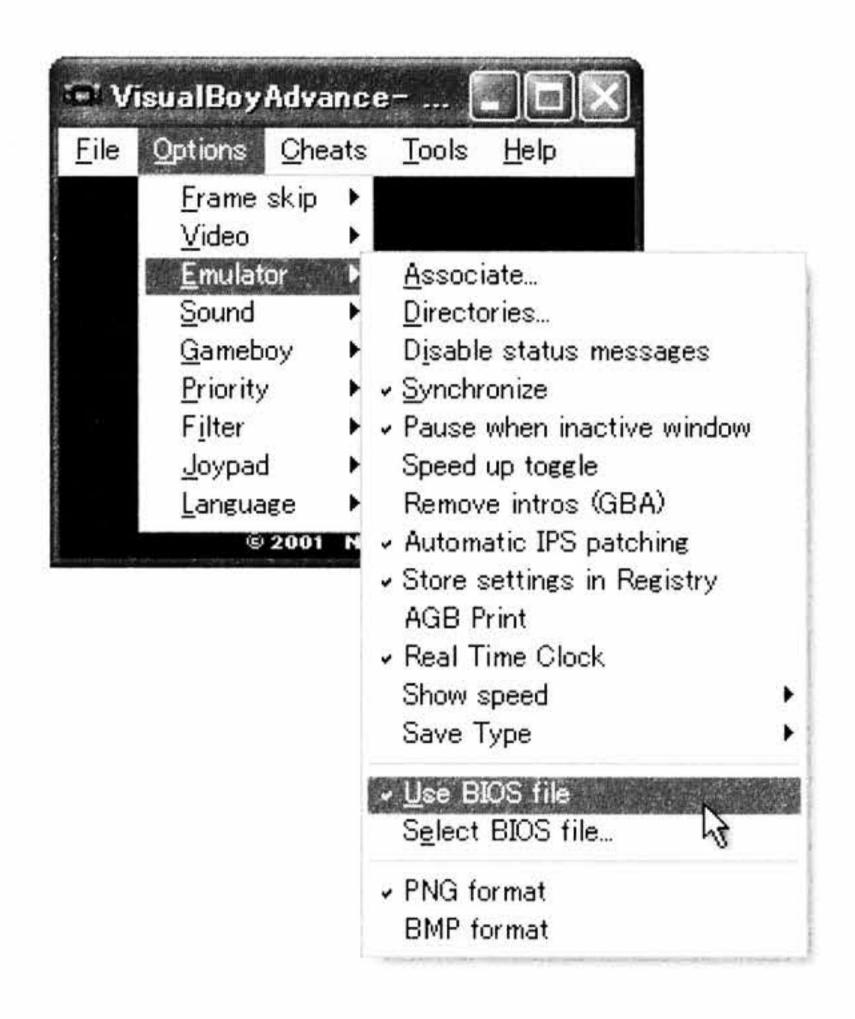
○カートリッジのバックアップメモリ内にはBIOSがコピーされています。カートリッジのソフトを単体で起動すると、バックアップ領域に保持されたBIOSが消されてしまうので注意してください。あとは通常のバックアップメモリのセーブと同様の手順で、内容をPCへ転送します。

注意!!

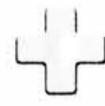
カートリッジのバックアップ領域にBIOSを書き込むので、バックアップ内容は破壊されてしまいます。必要であればあらかじめバックアップを取るようにしてください。

●手順6：BIOS の整形

○吸い出しサイズは利用するカートリッジの容量に依存します。今回利用するバックアップ容量は256Kbit（32KB）なので、吸い出したファイルサイズもこれと同様になります。実際のBIOSは先頭の16KBだけなので、不要な後半部分は切り落としてしまいましょう。

BIOSの設定

「Select BIOS file...」でファイルを設定します。ファイル設定後、VBAでのBIOSの使用設定は図のように行ないます。

BIOSの確認

VBAでカートリッジから吸い上げたソフトを実行させて、BIOSが設定されたかどうかを確認します。起動時に図のような実機同様のオープニングタイトルが出ればOKです。

GBAをPCのゲームパッドにする

ULA-GP

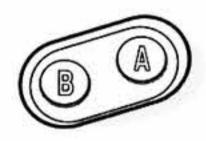
TeamKNOxでは元々、GBCの開発をこれまで手がけてきました。その中でも異色だったのがGB-CONです。これはGB自体をPCのゲームパッドに変えてしまうプログラムです。ULA-GPはGB-CONのGBA版と考えてもらってかまいません。ULA-GPはGB-CONに比べるとはるかに強力で柔軟です。ULAの名がついているのでおわかりかと思いますが、ここでもULAの技術を応用しています。ちなみにULA-GPのGPとはGamePadの略称です。

ULA-GP_V2で最も出色なのは、GBAのエミュをGBA自身で楽しむことができることでしょう。「実機並みの操作性」という言葉がありますが、実機で操作できるので、この言葉は意味を持たなくなります(笑)。

GameBoy Advance SPをゲームパッドにしてGBAエミュレータ「VBA」でReversi Advanceを遊ぶ

ULA-GPの動作原理

ここでULA-GPの動作シーケンスをトレースしてみましょう。つまりUSBの動作モードのおさらいです。USBの動作モードとしては、主なものだけでも「バルク」「アイソクロノス」「HID」などがあります。

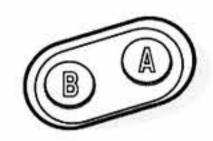
バルクはデータを正しく送るためのモードで、アイソクロノスは実時間の転送を主として行ないます。つまり、音楽や動画などのリアルタイム系の処理です。HIDはキーボードやマウス、ジョイスティックなど対人間用デバイスに用います。ULA-GPはバルクとHIDを連動させることで実現しています。

7-3-2-1 ULA-GPの動作シーケンス

最初にファームウェアがULA本体に転送されます。このファームウェアはGBAにプログラムを転送させるためのもので、ULA全般で利用されています。そのプログラムが転送されたあと、GBAをゲームパッドにするソフトが転送されます。その後、ULAをHIDにするファームウェアを転送します。まとめると次のようになります。

① ULA本体にGBAのプログラムを転送するファームウェアを転送。
② ULAを介してGBAをゲームパッドにするプログラムを転送。
③ ULA本体にGBAをHIDにするファームウェアを転送。

①～③でGBAはゲームパッドになっています。今回のファームウェアの設定では、16個のボタンと4方向、スロットルとラダーが設定されています。これらの設定はファームウェアによって行なわれます。スロットルとラダーにはダミーデータが入っていますので常に固定です。

ULA-GP_V2

例によってULA-GPは最初に作ったものなので、試作の色合いが強いプログラムです。同じ機能を作り直すのも能がないので、新機能を追加したULA-GP_V2を制作することにしました。

7-3-3-1 ULA-GP_V2の特徴

① ULAの新型クラスライブラリを利用

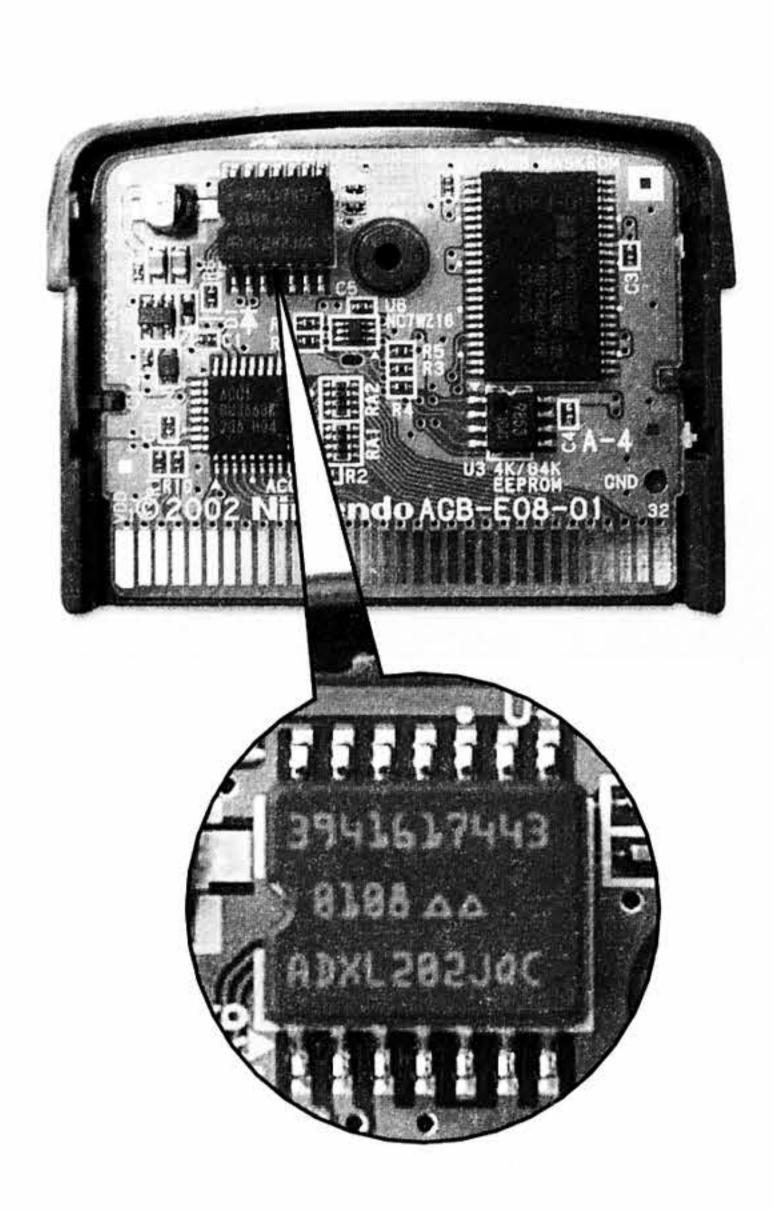
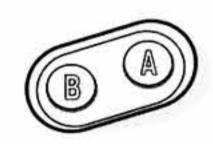
これにより、共通したクラスライブラリを用いることが可能になりました。

② GBACCにより、GBA側プログラムの小型化

ULA-GPではDevkitAdvの各種ライブラリを組み込んで使っていたため、プログラムサイズが64KBとかなりの大きさでした。TeamKNOxLib.hの利用により機能を増やして、プログラムサイズを小さくすることができるようになりました(たったの9KB!!)。

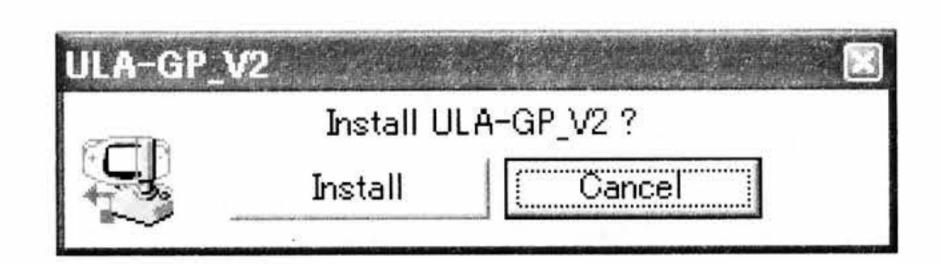
③ ハッピィパネッチュのモーションセンサーの利用

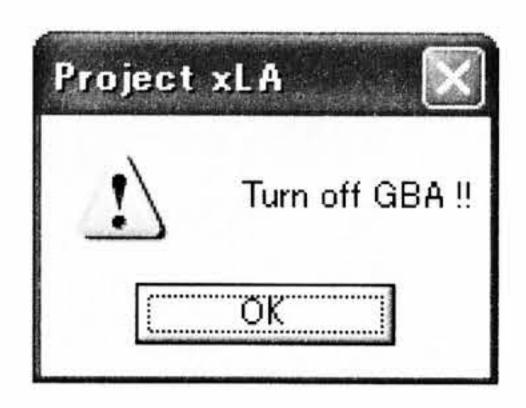
ブートモードの活用により、カートリッジの付加機能を利用できるようになりました。今後、さまざまなセンサが搭載されることが予想されますが、それのアクセス方法がわかればULA-GP_V2に付加機能として取り込むことが可能です。

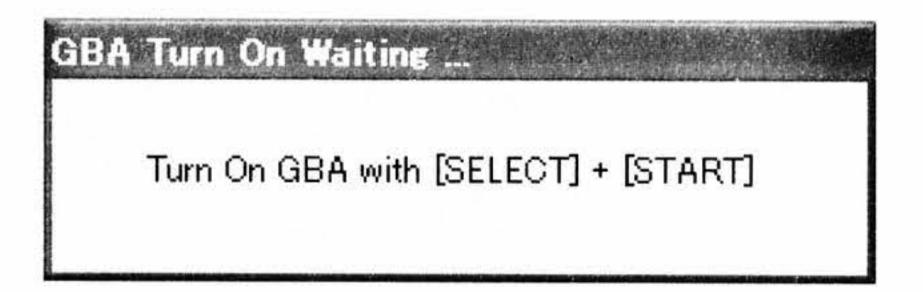
「コロコロパズル ハッピィパネッチュ！」のカートリッジに搭載されたモーションセンサ。右はそのセンサ部分

7-3-3-2 ULA-GP_V2の動作シーケンス

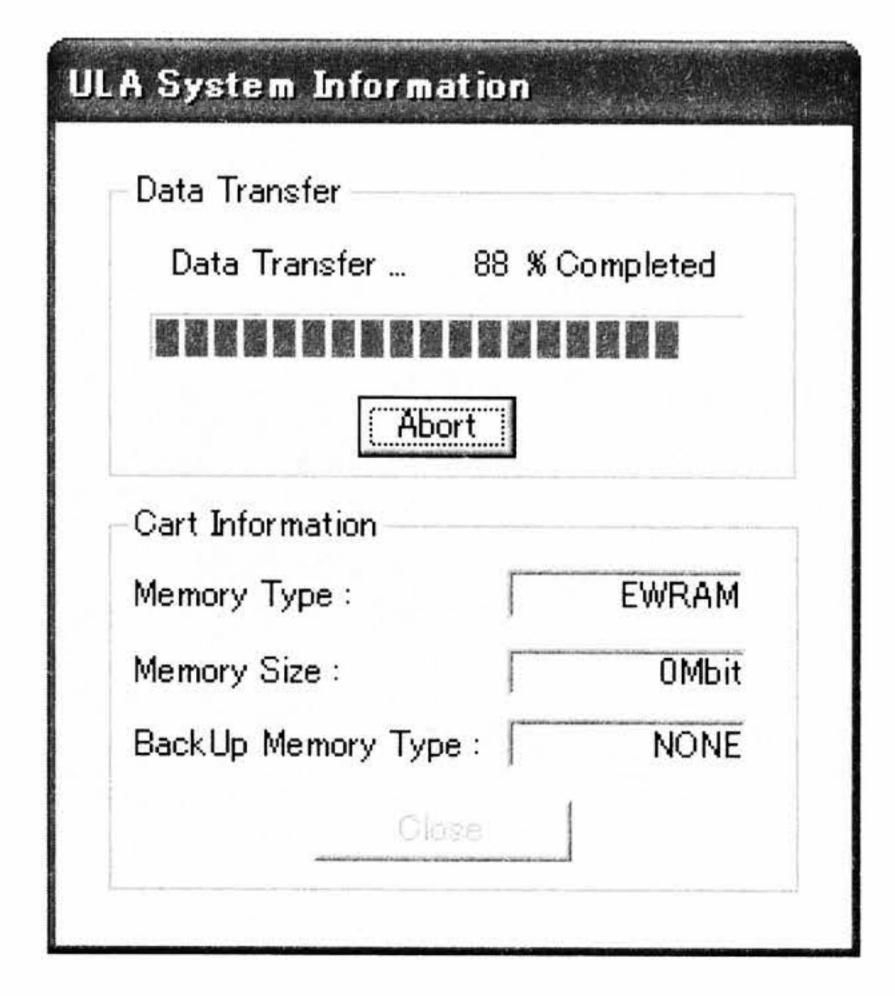
ここで、ULA-GP_V2の大まかな動作の流れを見てみましょう。

①起動直後

Installを押せばインストールに進み、止めたいときはCancelを押します。

② GBAの電源を切る

ここからは、新型ULAのクラスライブラリを利用しています。よって、ここからGBAのソフトのダウンロードまでは共通の操作になります。

③ GBAの電源投入

[START] [SELECT] を同時に押しながら電源投入することで、ブートモードにします。

④ GBA側ソフトのダウンロード

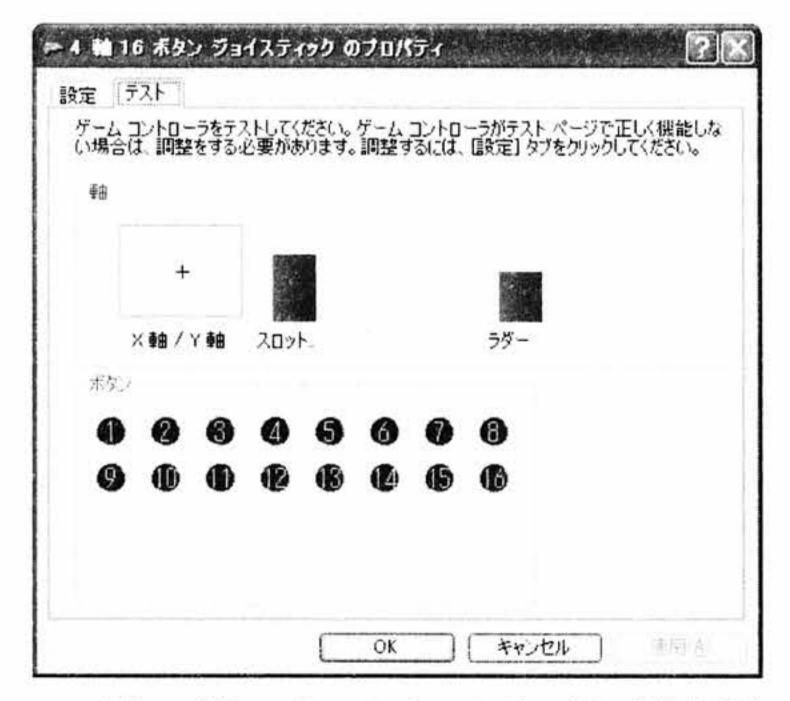
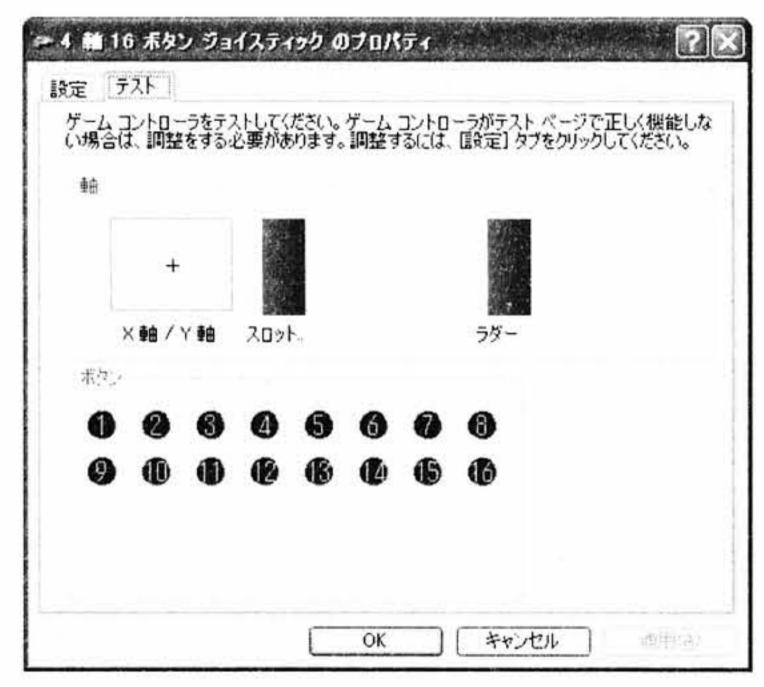
GBAソフトがダウンロードされます。

⑤ファームウェアのダウンロード

これは画面から読み取ることはできませんが、極めて重要な動作をしています。USBをHIDにするための動作を行なっています。

⑥ GBA側のソフトの動作開始

GBA側のソフトが動作します。Tht.とLad.の2つはそれぞれ、ハッピィパネッチュのセンサ出力に対応しています。

⑦動作の確認

ハッピィパネッチュのカートリッジが差し込まれている場合はスロットルとラダーが使えるようになります。

ハッピィパネッチュのカートリッジが差し込まれている場合

ハッピィパネッチュのカートリッジが差し込まれていない場合

GBAでファミコンエミュレータを楽しむ

テレビゲームといえばファミコンでしょう。もはや一般名詞です。筆者の友人の母親などは、セガのマスターシステムを指して「セガのファミコン」などと言っていたくらいです。このテレビゲームの市民権を確立したファミコンも現在では生産中止です。しかし、多数のエミュレータが開発されている現在、GBAでもこのファミコンのゲームをプレイすることができるのです。そうです、GBAのパワフルな処理能力は、任天堂をテレビゲームのメーカーとして不動の地位に導いたあのファミコンでさえエミュレート可能にしたのです。

GBAを手にした当初、筆者はGBAの開発はこれまでの延長線にはないような気がしてなりませんでした。それが、最近になってある種の確信が生まれるまでになりました。GBAのポテンシャルはGBCの比ではありません。筆者の答えとしては、GBAは過去に発売された各種エミュレータのプラットフォームとして最適なのではないか(?)ということです。

ファミコンエミュレータ

ファミコンエミュレータとしてリリースされたものは現在（2003年11月）のところ2種類です。開発中ではさらにもう1つあり、筆者も関わっています。その2種類のファミコンエミュレータの特徴を簡単に紹介しましょう。

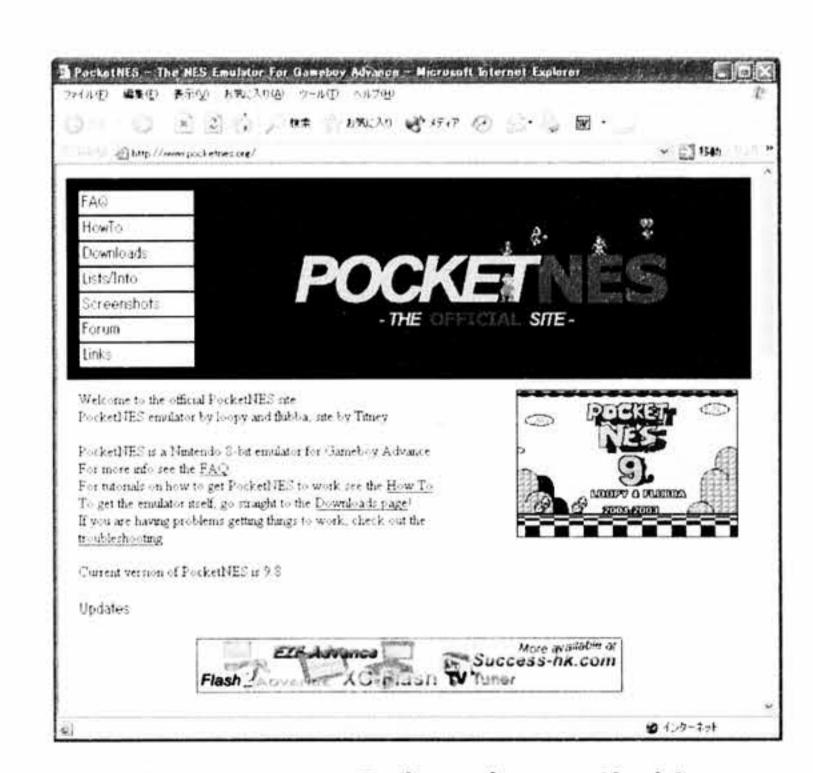
① PocketNES

GBA用の最初のファミコンエミュレータです。ソースが公開されています。エミュ自体かなり小型（40KB程度）なので取り扱いが楽です。

② ファミコンアドバンス

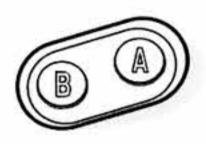
国産のエミュレータ。メニューなどが日本語になっていて使いやすい。

筆者はPocketNESを主に利用しています。

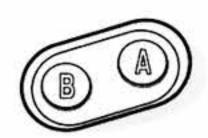
VBAで動かしたPocketNESのゲーム画面

PocketNESのオフィシャルサイト

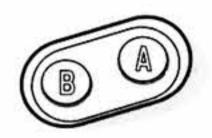
Thingy-ULA

Thingyとは、PocketNESの開発者でもあるLoopy氏が開発したROM結合ツールです。ROM結合ツールにはさまざまなタイプのものが存在しますが、筆者にとっては機能が豊富すぎてイマイチ使いにくいものでした。そこで作者のLoopy氏にコンタクトを取り、ソースを頂くことにしました。

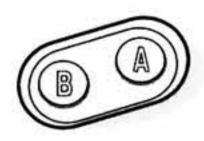
「Thingy-ULA」は基本的にROM結合ツールにULAの転送機能を付けたものです。つまり、自分で選んだROMイメージをそのまま転送することができます。また、結合した状態でのファイル出力ができるため、それらを保存しておけば、今後はそれらを転送することでファイル選択の手間を省くこともできます。ファミコンゲームのROMイメージは大変小さく、あの「スーパーマリオブラザース」ですら40KB程度です。つまり、内部RAMだけでもこの程度なら4本程度を格納することができます。

ゲームを選んで携帯する

GBAなどの携帯ゲーム機の最大の特徴は、何といっても携帯できることです。当たり前のことですが、これはかなり重要です(笑)。その他に携帯できるデバイスとしては、携帯電話やMDプレイヤー、MP3プレイヤーなどもありますね。

さて、MDやMP3プレイヤーでは、自分の好きな曲をあらかじめこれらのプレイヤーに仕込んでおいてから、外に持ち出して(携帯して)利用します。同じことがGBAでもできないかと思ってチャレンジしたのが、このThingy-ULAです。

GBAでエミュのゲームを楽しむには… ROM結合ツールについて

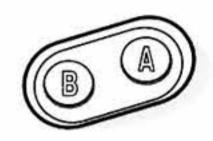
話が前後しますが、前記のエミュレータをGBAで楽しむには「ROM結合ツール」が必要です。これはGBA単体にはWindowsやDOSのようなファイルシステムが存在しないためで、ROMイメージをファイルに見立ててGBAからアクセスする仕組みです。エミュレータ本体と複数のターゲットゲーム機のROMイメージを結合して、1つのファイルを作成します。このときに作成されたファイルは、GBAの実行ファイルになります。このROM結合ツールも各種出ているので、自分の好みでチョイスすればいいでしょう。

ファイルを転送する

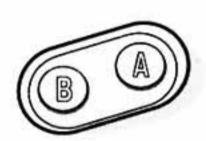
作成したファイルは、GBA用のフラッシュカートリッジに書き込みます。通常は書き込みハードとしてULAが、書き込みソフトではULA-HostV2やFlashManagerなどが使われています。

ここまでに、2種類のステップが存在しています。「ROMイメージの作成」と「ファイルの転送」です。慣れれば問題ないですが、いずれにしても面倒です。もう少し簡単にできないものかと思ってしまいます。また、フラッシュカートリッジはそれなりに高価です（容量にもよりますが、両方はセットで15,000円くらいはするでしょう）。

もっと安く

お金をかけずに安くできないか（?）も検討してみましょう。

PocketNESを詳しく見てみると、内部RAM（マルチブート）だけで実行できることがわかります。つまり、この領域にプログラムを転送して楽しむことができるわけです。

ただし、RAMなので電源をOFFにすると内容が消えてしまいます。内部RAMにPocketNESを転送して楽しむためには、GBAの電源をずっとONにしていなければなりません。しかし、この辺についてもPocketNESはよく考えられていて、CPUのクロックを停めて消費電力を下げるSleepモードが用意されています。この機能により、何もしなければ電源をONにしたままでも3～4日間は電池が持ちます(これはプレイ時間ではありません。念のため…。電源ランプが緑をキープしているだけです)。

もっと簡単に…

MP3プレイヤーの曲の転送のように簡単にできないかと考えます。幸い、筆者の手元にはULAやそれに準じたシステムがあります。PCからGBAへの転送はこちらを使えばよさそうです。

ここまでのことをまとめると…

「ファミコンエミュレータとファミコンのROMイメージを結合してGBAの内部RAMへ転送する」ことを実現するソフトができれば、GBAで安く、簡単にファミコンのゲームを楽しめるわけです。

STEP1：ROMイメージの作成

PocketNES
ROM image
ROM image

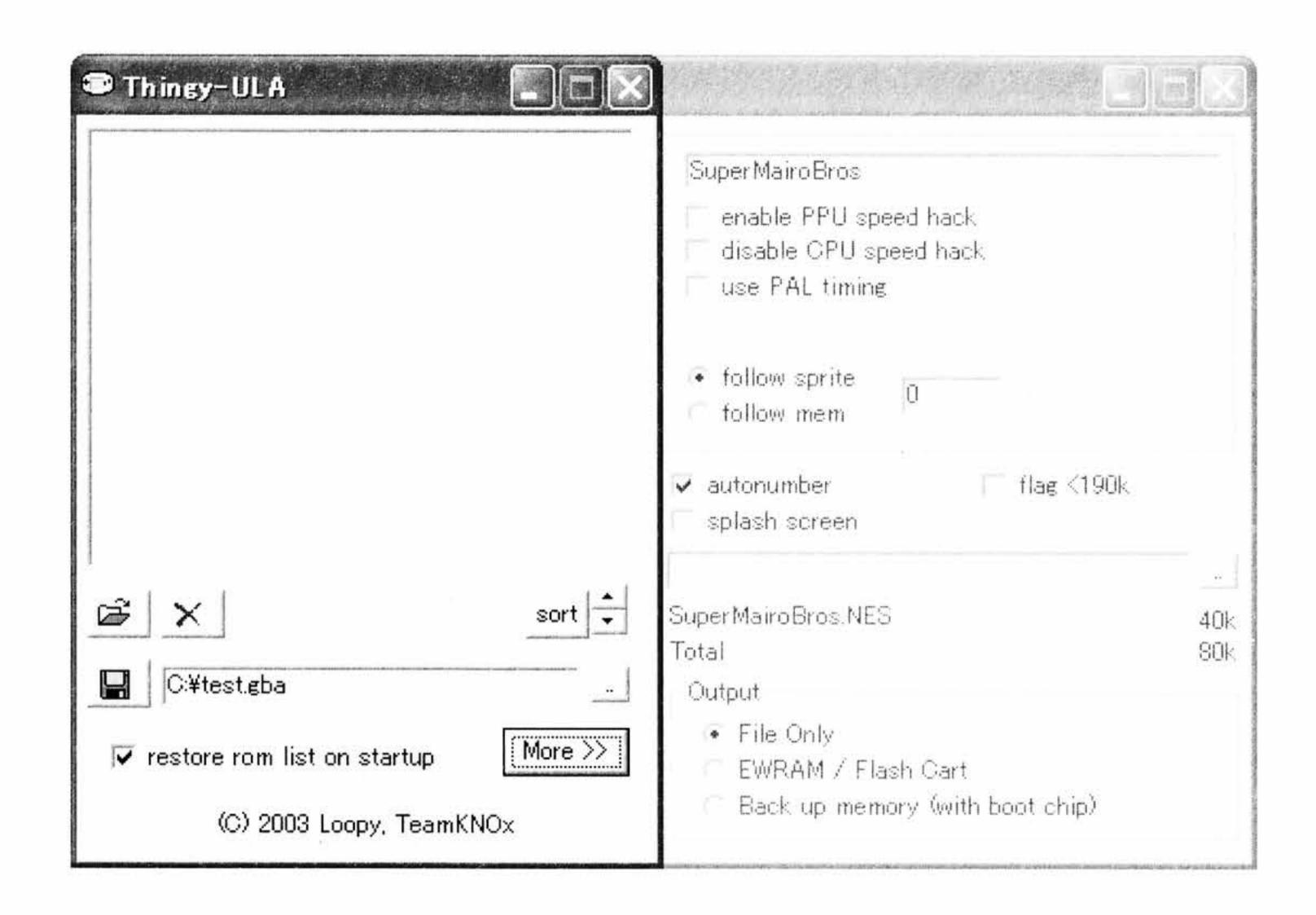
①ROMイメージの結合

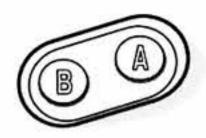
②ROMイメージとエミュレータの結合

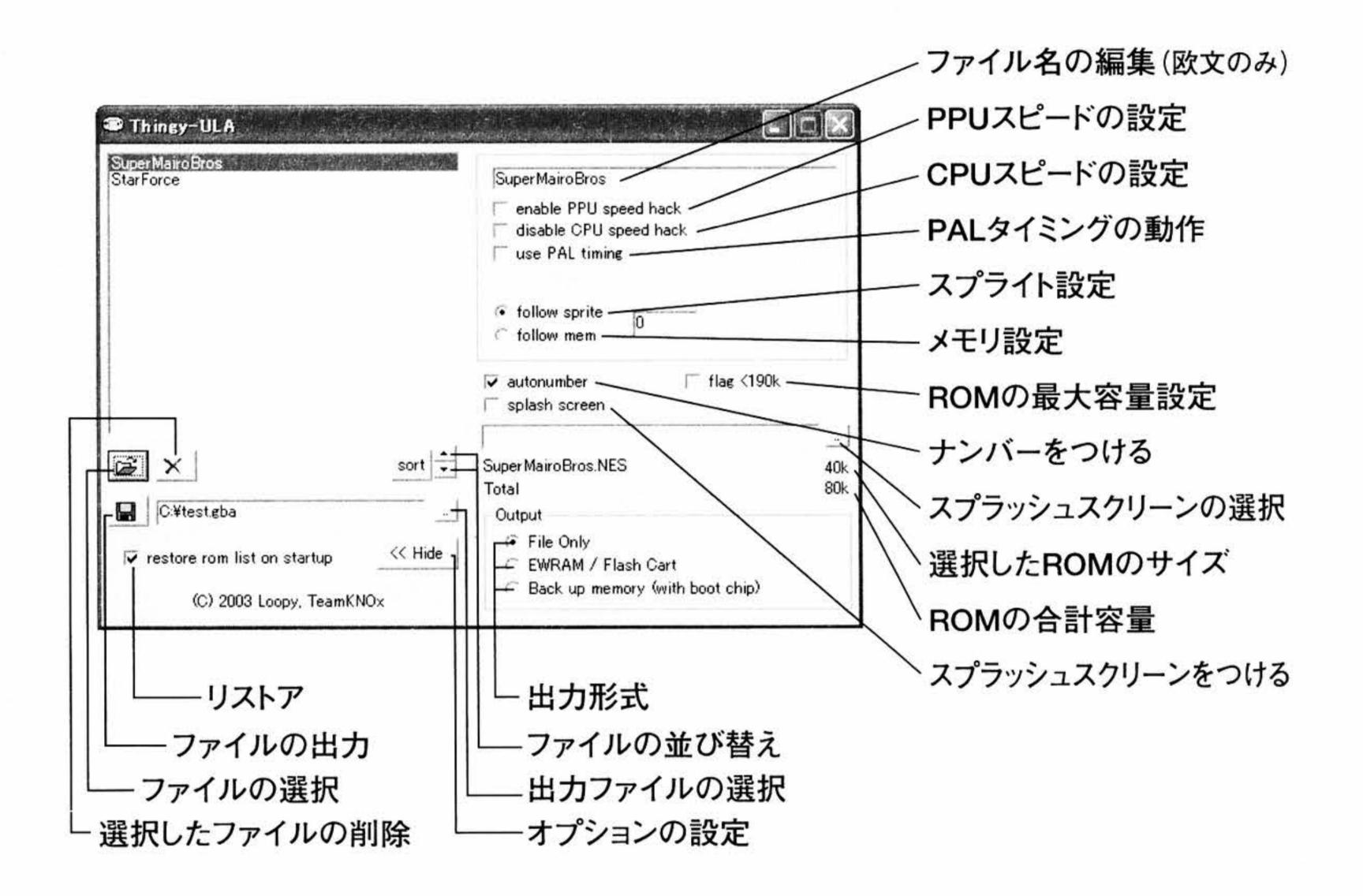
STEP2：ファイルの転送

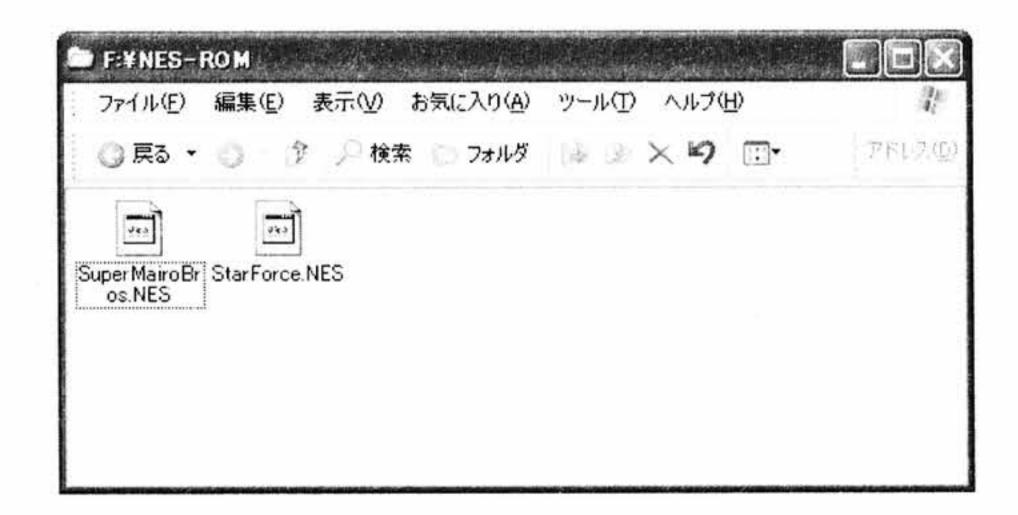
③本体RAMに転送

Thingy-ULAの使い方

起動すると下図のようになります(濃い部分)。[More] ボタンをクリックすると、各種設定を行なう画面が右側に表示されます(薄い部分のこと)。

初期画面はシンプルなリスト画面のみです。リストへの登録は [..] ボタンでROMイメージの保存してあるディレクトリを指定する方法のほか、ドラッグ&ドロップにも対応しています。

[More] ボタンをクリックすると設定画面を開きますが、右下の出力オプション (Output) が追加したULAの機能です。

7-4-3-1 基本的な使い方

Thingyの使い方はとても簡単です。まず、基本的な動作を見てみましょう。

● ROM イメージファイルの選択

吸い出したROMイメージファイルを左側部分にドラッグ＆ドロップします。あるいはフォルダボタンをクリックして、通常のファイル選択ダイアログからファイルを選択します。

●選択したROMイメージファイルの削除

ファイルを選択（ファイルネームをクリック）したあと、[X] ボタンをクリックします。

●ROMイメージファイルの並び替え

ファイルを選択した後に[▲][▼]をクリックします。

●出力ファイルの選択

結合したファイルを出力します。このとき注意しなければいけないのが、既にあるPocketNES.gbaに上書きしないようにすることです。上書きしてしまうとPocketNESファイルが破壊されます。

●オプション設定ボタン

右側のオプション設定を表示する場合はこれをクリックします。

●オプション設定

左側はThingy-ULAを使うためには必須の設定です。右側はより進んだ使い方をする人向けです。通常はデフォルトの設定で問題ありません。ULAを使う場合はOutputの設定のみで問題ないでしょう。

●ファイル名の編集

ゲーム選択時の名前を付けることができます。ただし、使用できるのは欧文のみです。ひらがな、カタカナ、漢字は使えません。

●PPUスピード設定

Picture Processing Unitのスピード設定をします。

●CPUスピード設定

Central Processing Unit のスピード設定をします。

●PALタイミングでの動作

PAL向けに作られたROMではこちらを設定するといいでしょう。

●スプライト設定、メモリ設定

それぞれ、実機に近づけるため、あるいはより快適にプレイするための設定です。

●ナンバーをつける

各ゲーム名の先頭にナンバーをつけます。

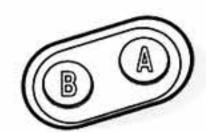
●スプラッシュスクリーン

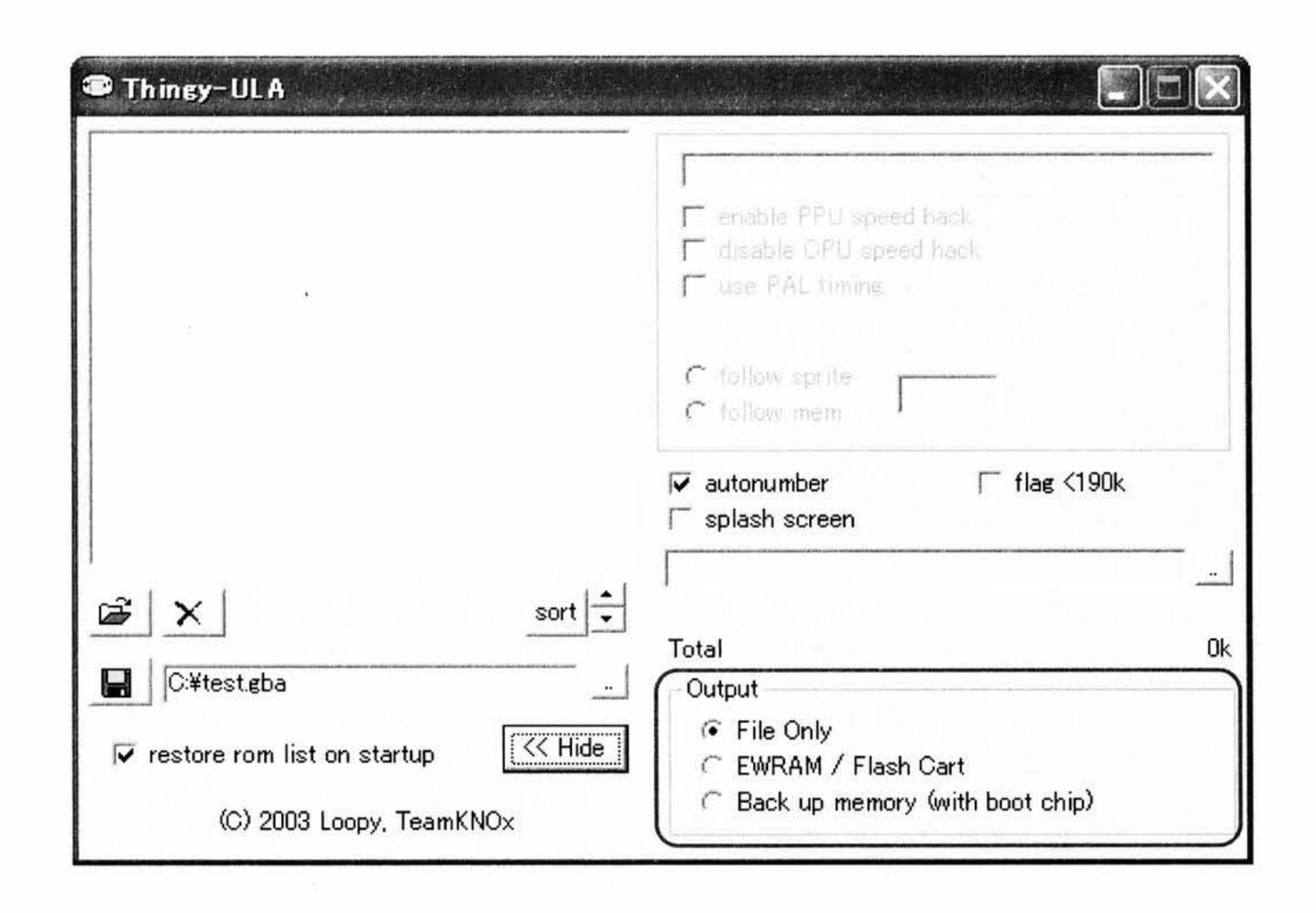
PocketNES 起動時の画面（スプラッシュスクリーン）を出します。240×160 のビットマップファイルを設定します。

●選択したROMのサイズ

選択したROMのサイズが表示されます。

●合計のROM容量

結合しようとしているROM容量の合計が表示されます。カートリッジの容量と勘案するといいでしょう。

7-4-3-2 拡張部分の使い方

Thingy-ULA はオリジナルの Thingy を拡張するカタチでできています。右下段が拡張したオプションです。拡張した部分は ULA のクラスライブラリの動作に対応しています。

● File Only

これはオリジナルと同じ動作です。PocketNES と NES の ROM ファイルを結合して *.gba ファイルとして出力します。エミュレータでプレイしたい場合や、他に使い慣れた転送ツールを使っていて *.gba ファイルだけを出力したい場合はこのオプションを選択してください。

● EWRAM / Flash Cart

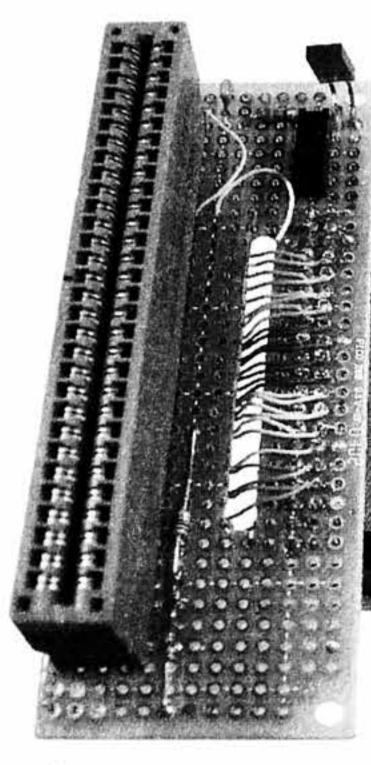
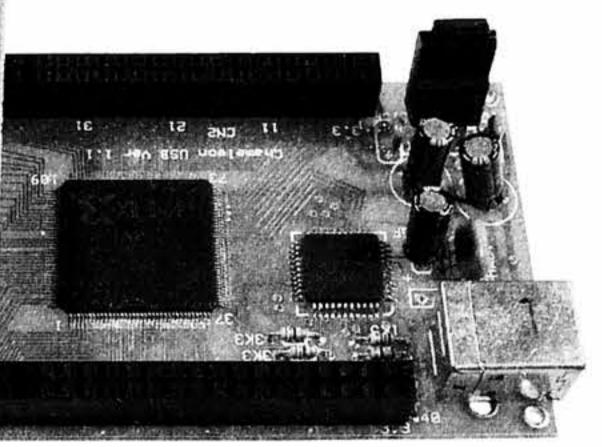
GBA のワークRAM (EWRAM) か Flash Cart にファイルを転送するためのオプションです。

● Back Up Memory

ブートチップに対応するためのオプションです。カートリッジのバックアップエリアに書き込みます。

Column

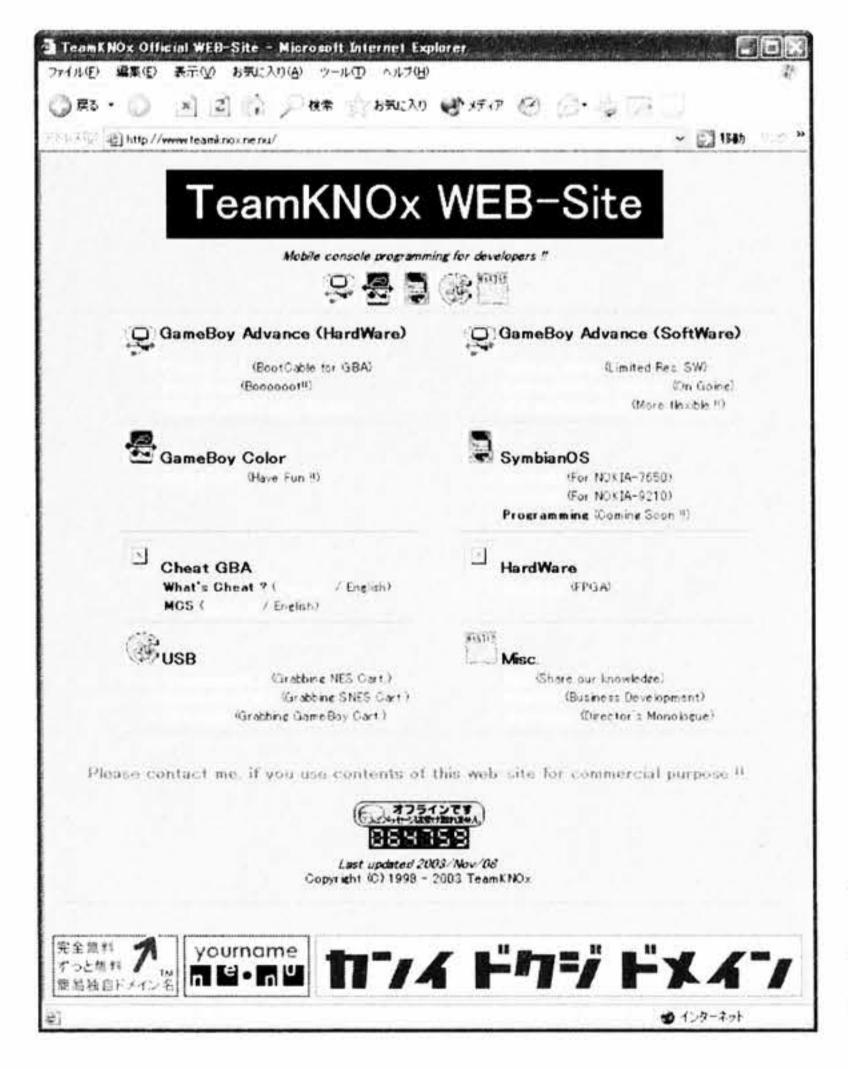
カートリッジの吸い上げ

筆者自身は法律の専門家ではないので、技術的な側面だけを述べておきましょう。ファミコンのROMイメージファイルをどの様に入手するかですが、P2Pによるファイル交換システムがほぼ確立している現在、こちらで入手することは造作もないようです。ただし、「技術者たるもの自分の力でなんとかしたい」ということで、吸い上げシステムを構築してみました。ブートケーブルでおなじみのOptimize氏からリリースされているカメレオンUSBを用いて吸い上げ機を作ってみたのです。本書の内容から逸脱するので詳細は掲載しませんが、再現性も高いのでぜひ挑戦してみてほしいと思います。

Chameleon NES（Chameleon USB＋ファミコンカートリッジ装着アダプタ＋ソフト）**：左側の大きなコネクタにファミコンカートリッジを装着します**（詳細はTeamKNOxのHPで公開）

TeamKNOx

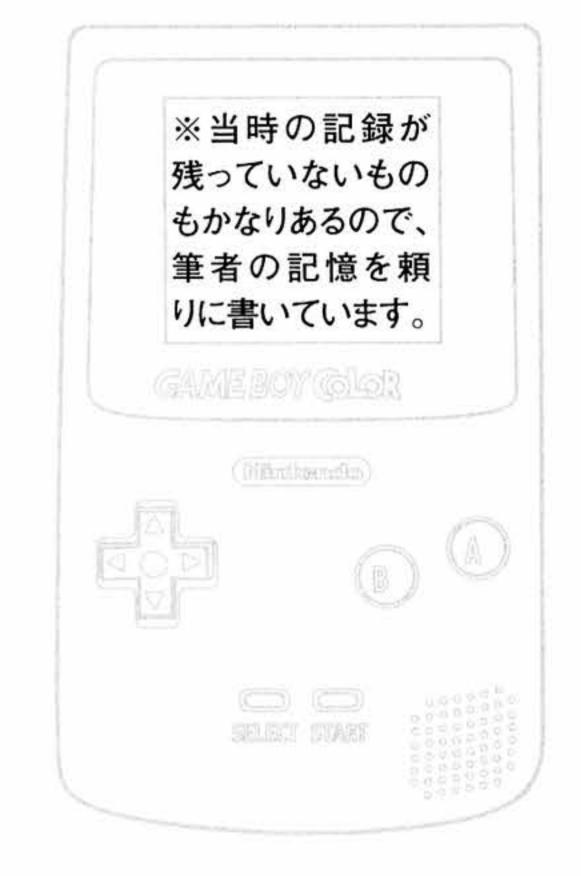
本書の「あとがき」の代わりに、筆者が所属しているTeamKNOxとその活動をとおして感じた筆者の考えを簡単に紹介したいと思います。これから本格的にプログラミングをはじめようという読者の参考になれば幸いです。

TeamKNOx 活動の歴史

自分が所属している団体なのですが、極めて不思議な団体です。別にお金がもらえるわけでもないし、たくさんのインセンティブ（ご褒美）があるわけでもありません。ただ、開発はまじめにやっていますし、プログラミングやモノ作りの楽しさを広く啓蒙したいと日夜活動しているのは事実です。プログラミングは孤独な作業です。あなたのプログラミングライフの何か参考になるかもしれないと、ここでは少しTeamKNOxの活動を紹介したいと思います。

TeamKNOxのホームページ
●GBC/GBAに関する情報を中心に扱っています
URL http://www.teamknox.ne.nu/

TeamKNOxの活動史

1995　　: TeamNOx結成（このときは N氏と大橋が2人で立ち上げた）。Mac用の各種通信システム製作・実験
1996　　: PC-NCU, TAXなどの実験
1997　　: 本格的にGB関連の開発に着手（新N氏参加、旧N氏は音信不通となる），DSCAN, GB-DTMFを発表。GBDKのサンプルコードになる
1998 / 02 : GBDKの作者Pascal Felber宅訪問・滞在
1998 / 07 : TeamNOxからTeamKNOxに名称変更。ROM解析のエキスパートReiner Zieglerとミュンヘンで邂逅
1999 / 04 : 初めてのGBCソフトColorBar, DSCANの開発
1999 / 05 : 赤外線の魔術師K.I.氏「TeamKNOx」に加入。GBCによる赤外線学習リモコンの制作に着手
1999 / 06 : DATEL社TeamKNOxメンバーをU.K.本社に招待
1999 / 10 : 本格的ゲームNamiheyの開発
1999 / 12 : Lego MindStorms用リモコンRoReCon開発
2000 / 04 : GB-PDAの開発、本格化
2000 / 06 : GB-KEY開発
2000 / 07 : GB-fREMe開発
2000 / 09 : GB-Term開発
2000 / 11 : GB-BAR開発
2001 / 01 : GB-CAT開発
2001 / 02 : GB-PDC開発
2001 / 03 : 「ゲームボーイのプログラム・改造マニュアル」出版。GB-GPS開発
2001 / 05 : Pelmanism開発
2001 / 10 : USB-GBX開発。GB-CON開発
2001 / 11 : GB-PROPO開発
2001 / 12 : USB-Linker開発
2002 / 01 : USB-Meeting 2002開催

2002 / 02 : GBAプログラム開発開始
2002 / 03 : ULA開発開始
2002 / 05 : ULA-GP開発開始
2002 / 06 : Treva-GBA開発
2002 / 10 : 「ゲームボーイプログラミング入門」出版
2002 / 11 : ReversiAdv開発
2003 / 04 : ULA-HostV2開発開始
2003 / 05 : ZeroMineAdv.開発
2003 / 07 : ゾイドコントローラ

ビジネスモデルとプレイモデル

経済誌や最近ではPC雑誌なんかを読んでいても「ビジネスモデル」という言葉が出てきます。これは仕事のやり方を定義したものと言い換えることができます。たとえば「インターネットを使った新しいビジネスモデル」のように使ったりします。インターネットを使うことで新しい仕事のやり方が生まれたように、新しい遊びのやり方も生まれていいはずです。それがTeamKNOxが提案する「プレイモデル」です。といっても具体的な何かがあるわけではなく、知識や面白いことを共有することがメインのモデルです。TeamKNOxはゲームボーイ(アドバンス)の開発で、その筋(?)では少し有名になってきました。これはTeamKNOxのポリシー(プレイモデル)によるところが大きいと自負しています。

そのモデルとしては…

- 楽しくやる
- お金をかけない
- 共有する

せっかく、高速PC(ってよく言われるけど、マジでひと昔前のスーパーコンピュータと同じくらいの性能なんですよ)やインターネット、楽しいゲーム機が出てきたのだから、みんなで色々遊べたら面白いと思うわけです。ここで、少しモデルの説明をしておきましょう。

楽しくやる

趣味に没頭している人間は子供みたいなものです。時間を忘れて熱中します。つまらなければやりません。だいいち、楽しくなければ長続きしません。TeamKNOxの活動も楽しくやることを大切にしています。

お金をかけない

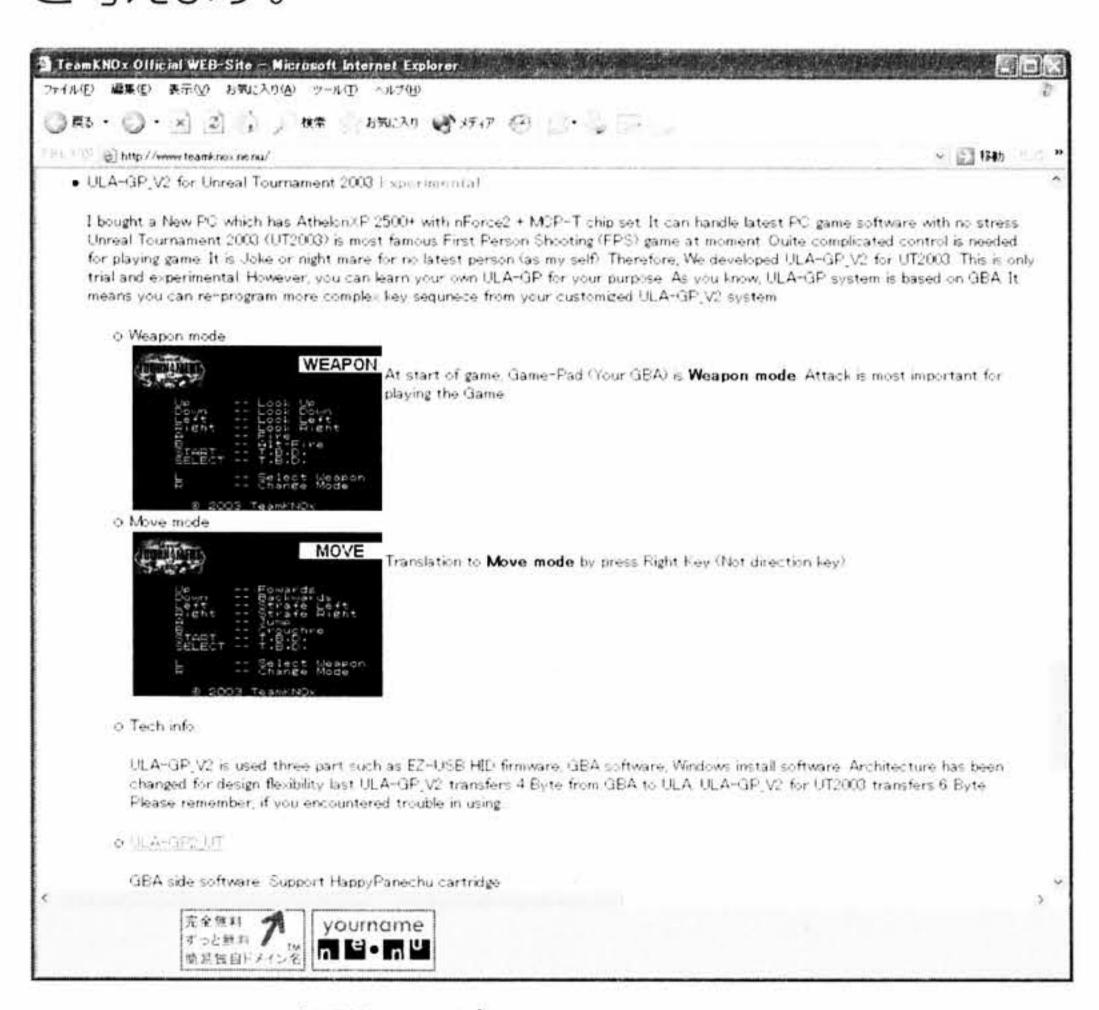
活動自体が何か見返りを求めて行なわれているわけではないので、個人でかけられるお金にも限りがあります。ただ、機材やサンプルなどの寄付は大助かりなので積極的に受け入れています。製品評価の場としても利用してもらっています。

共有する

共有というのが今後の重要なテーマになると思います。いろいろと物議を醸しているP2Pの「ファイル共有」なんかは顕著な例ですね。一方で与える側は神と崇められたりします。面白いですね。

TeamKNOx的にはあくまでも「共有」をしたいと考えています。価値観の共有、知識の共有、面白さの共有を国境や年代や性別を超えて進めたいのです。良いものができたら、みんなに「見てもらいたい」「使ってもらいたい」「より良くしたい」と思うのは開発者としての本能だと思います。これを実現するには、なるべく多くのリソースの共有が必要です。共有によって、それぞれ得意な分野の人たちによる「改良」が可能になるのです。

TeamKNOxのWebページはなぜ英語なのか？

最近になって、日本のアニメやゲームは世界に誇るコンテンツだともて囃されています。しかし、それ以外のものについてはあまり理解されていないのが現状です。そこで、GBAやケータイなど、日本が世界に誇るコンテンツをさらに広く深く世界中の人に知ってもらいたい、また、GBAに興味を持つ日本人にも、もっと世界を意識してもらいたい、そんな理由から英語で書いています。実際のところ、興味があることなら英語はたいして障壁にならないでしょう。むしろ、メリットのほうが大きいのではないかと考えます。

ULA-GP_V2の解説ページ（中学英語レベルで充分理解できると思います）

最近、気になっていること

フリーウェアとオープンソース

TeamKNOxは原則として、ソースも含めてフリーで公開しています。少なくともGB関連はすべてオープンソースです。これらを使って何をやるかは問いませんが、利用しているのならひとことくらい挨拶が欲しいところです。それで、利益を上げているのならなおさらです。吸出し機能を持った国産のファミコンエミュレータなどは、我々が開発したノウハウを利用しているようです。交渉が面倒なのはわかりますが、他人が時間と労力をかけて作ったものを利用する限り、何らかの挨拶があって当然だと思います。世知辛い世の中です(涙)。

ゲーム脳と手作りゲーム

「ゲーム脳」という言葉がマスコミを賑わしたことがありました。ゲームをたくさんやって大きくなると……、ゲームばっかりやっている若者の脳波は……、なんてことが意見されていたと記憶しています。しかし、これが話題になったのは、旧世代の本能的なTVゲームへの嫌悪感の琴線に触れたからだと思います。新聞の子育てコーナーには「ウチではTVゲーム

の類はやらせないようにしています!!」とか、「ウチの子はやりたいと言ったことがありません」なんてことが書いてあってタメイキが出ます。

筆者が出版社やメーカーからゲームのサンプルを頂戴する機会が多いことも理由のひとつではありますが、基本的に筆者の家庭ではゲームの類を禁止したりはしていません。むしろ推奨しているくらいです。

子供たちはゲームを漫然とプレイするだけでなく、必勝法はないか、そこに法則性はないか、いろいろと考えながら遊んでいるようです。これなら、ゲーム脳というのもあながち悪い意味ばかりではないなと思うのです。

また、子供たちを見ていると節度をもって遊んでいるようです(ちょっと、親バカモードに入ることをお許しください)。最近は子供たちの楽しみもたくさんあります。野球、サッカー、カラオケ、ケータイ、おしゃべり、読書、映画鑑賞、音楽……。ゲームはそのなかのひとつでしかありません。やるべきこと、やりたいことはたくさんあります。その楽しみのひとつを自分が知らないという理由だけで取り上げるのはいかがなものかと考えているわけです。

本書で紹介した「リバーシ」というゲームのルールは、小学生以上なら誰でも知っているものでしょう。ゲームをプログラミングするということは、このルールを詳細に分析していくことでもあります。これもまたゲーム脳の作業だといえるでしょう。本書付属のCD-ROMにはサンプルとして、デジカメで撮影した写真(BMPで画面サイズをGBAの画面サイズに編集してありますが)をGBAのディスプレイに表示するプログラムが収められています。これを利用して、子供の写真をオープニングやゲーム終了時に表示する新バージョンの制作はむずかしくないでしょう。その手作りゲームをプレイした子供はどんなことを感じるでしょうか? これも、また子供のゲーム脳を良い方向に伸ばすきっかけになるかもしれません。ぜひ、試してもらいたいと考えています。

日本の未来

以前筆者は若者の理工系離れを危惧した文書を書いたことがあります(『ゲームボーイのプログラム・改造マニュアル』データハウス刊)。最近、それに輪をかけて危惧しているのが日本のモノ作りそのものです。

筆者自身は仕事柄、各種の展示会によく行きます。先日も組み込み系システムの展示会に行きました。日本の有名メーカーの多くが出展していました。高性能マイコンチップの展示です。これらを用いた製品も多数展示されていました。最近流行のDVDレコーダなども目白押しです。多くの企業がともに素晴らしい機能を搭載した機種を投入しています。現在の売れ筋はこのDVDレコーダとケータイ電話だそうで、各社とも多くのリソースを投入して開発に当たっています。ただ、これらの製品には共通の基盤(プラットフォーム)がないために、各社バラバラの仕様で開発・設計が行なわれています。まるで、20年前の8ビットパソコンのようです(カンブリア紀の生物の進化のようにも思えます)。最終的にはPC98あるいはPC互換機のような進化の最終形で仕様が落ち着くのかもしれません。

筆者が危惧しているのはこの進化の最終形を日本のメーカーが出せるのかということです。PCではIBM-PC/AT互換機が製品としてすっかり定着し、その中で使われているCPUでインテルが、OSでマイクロソフトが世界を制しました。これと同様のことが起こって、結局日本のメーカー各社は単なる工場の役割で終わってしまうのではないかと想像してしまうのです。モノ作りも大切ですが、その先にあるもののビジョンを正確に捉えなければならない時期に来ているのではないでしょうか?

最後に

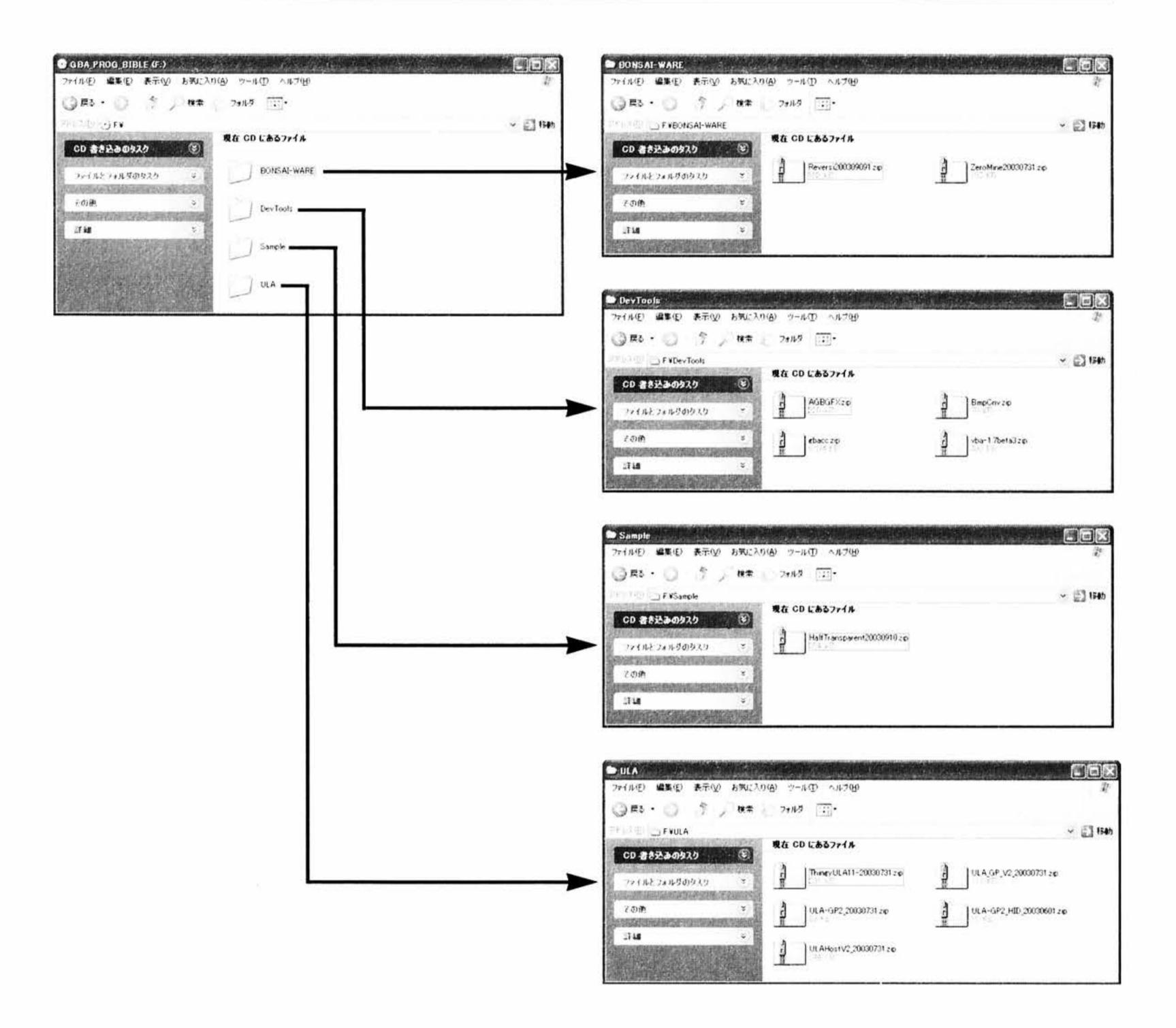
性懲りもなく３冊目のGB開発本を出してしまいました。

●謝辞

本企画を快諾してくれた担当のS.T.氏、TeamKNOxのK.I.、N.U.両氏、原稿完成まで励ましてくれた元西新宿３丁目、多大なアドバイスをくれた同僚のA.K.、K.K.両氏、テストプレイや写真撮影で協力してくれた長女・詩織、長男・零に感謝します。また、ULAをサポートしてくれているサポーター、そして最後まで読んでいただいたあなたにも感謝します。ありがとうございました。愛しています。

●参考文献

ARM プロセッサ　Steve Furber著／アーム(株)監訳……CQ出版社
インターフェイス　2002年11月号……CQ出版社
トランジスタ技術　2003年6,7,8月号……CQ出版社
モバイルプレス……技術評論社
月刊ゲームラボ……三才ブックス
ゲームボーイのプログラム・改造マニュアル……データハウス
ゲームボーイプログラミング入門……イーストプレス
bit別冊・ゲームプログラミング……共立出版
UNIX USER　2003年6-9月号……ソフトバンク パブリッシング

付属 CD-ROM について

CD-ROM に収録してある各ファイルは ZIP 圧縮してあります。Windows XP ではダブルクリックするだけで開くことができます。038 頁を参考に、必要なディレクトリ（フォルダ）に解凍したファイルをコピーしてください。WindowsMe/98SE/98/95/2000/NT4 では、ZIP ファイルを解凍（正確には伸張といいます）するソフトが必要です。WinZip や Lhaca 等の圧縮・解凍ソフトを Vector（http://www.vector.co.jp/）などで入手してください。

〔著者略歴〕

大橋 修（おおはし おさむ）

◎―1966年東京都墨田区本所生まれ。東京都立工業高専電気工学科卒。Apple IIの時代からコンピュータプログラミングに取り組んでいる。本業は主に組み込み系のシステム、アプリケーション開発。CPUアーキテクチャ、プログラミング言語、ネットワークなどのコンピュータ全般やロボットなどにも興味を持ち日夜研究している。得意分野はリアルタイム制御。

◎―主な著書に「究極MP3 & CD−R」(ソフトバンクパブリッシング)、「ゲームボーイのプログラム・改造マニュアル」(データハウス)、「ゲームボーイプログラミング入門」(イーストプレス)がある。

【TeamKNOx】 http://www.teamknox.ne.nu/

【編集】 WAKARU (bookworkshop@wakaru.co.jp)

ゲームボーイアドバンス・プログラミングバイブル

2003年 12月28日　　第1刷発行

著　者――大橋 修（おおはしおさむ）

発行者――八谷智範

発行所――株式会社すばる舎

〒170-0013 東京都豊島区東池袋3-9-7

東池袋織本ビル

TEL　03-3981-8651（代表）

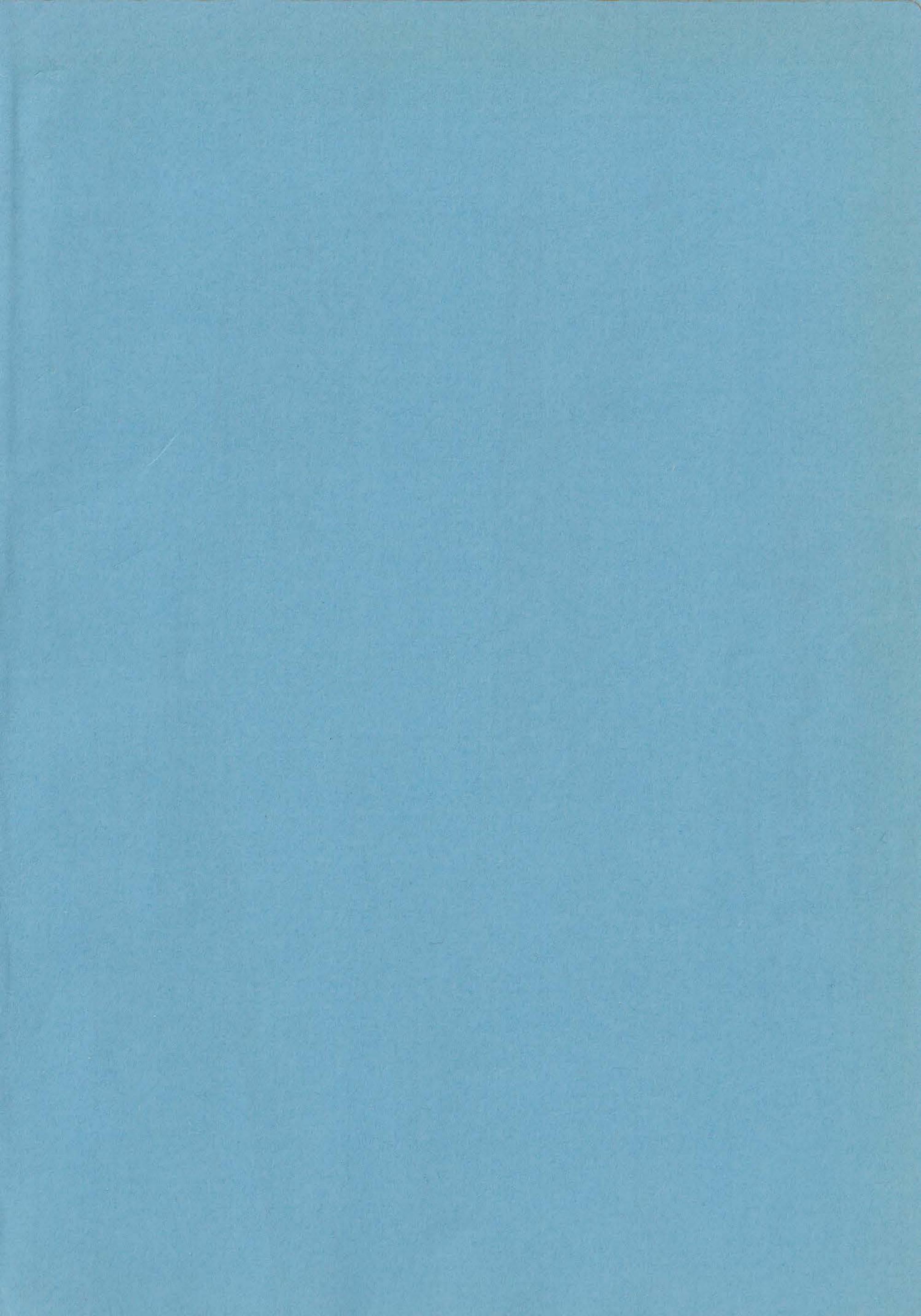
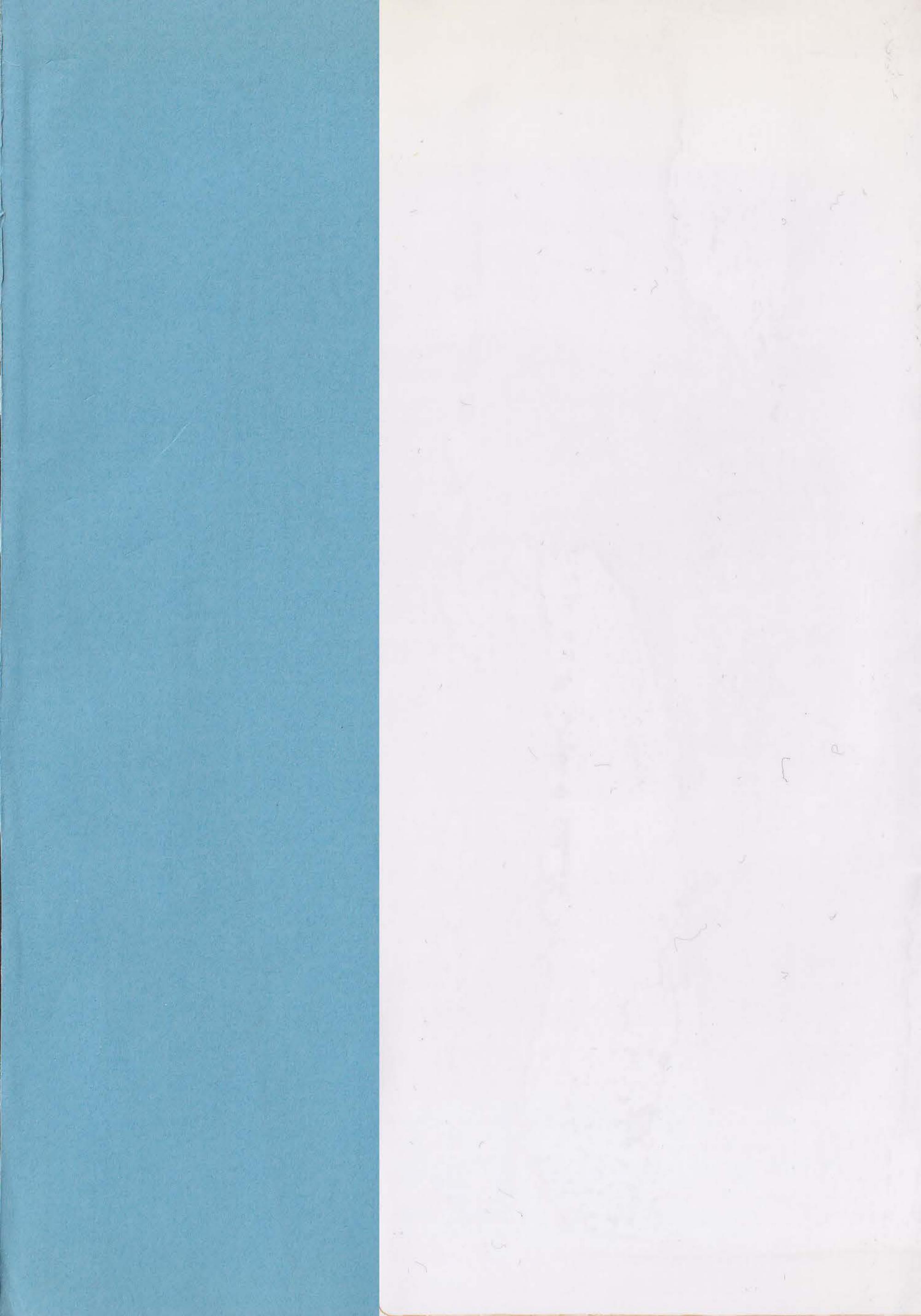
03-3981-0767（営業部直通）

FAX　03-3981-8638

振替　00140-7-116563

印　刷――図書印刷株式会社

ISBN4-88399-326-4 C0055　2003 Printed in Japan

GameBoy Advance Programming Bible
ゲームボーイアドバンス
プログラミングバイブル
大橋 修 著
Ohashi Osamu
Appendix CD-ROM
Subarusya すばる舎
COMPACT disc
for Windows
SELECT
START

ISBN4-88399-326-4

C0055 ¥2800E

9784883993260

定価: 本体2800円 (十税)

すばる舎

1920055028004

GameBoy Advance Programing Bible